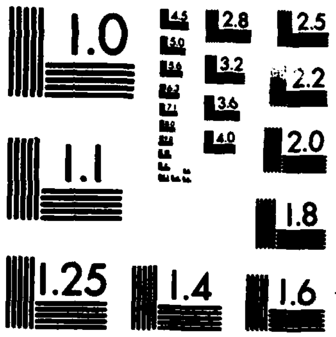
AD-A181 408 KNOWLEDGE BASED SYNTHESIS OF EFFICIENT STRUCTURES FOR 1/1
CONCURRENT COMPUTAT..(U) KESTREL INST PALO ALTO CA
R M KING ET AL. 31 DEC 86 KES-U-86-11 AFOSR-TR-87-0791
UNCLASSIFIED F49620-85-C-0015 F/G 12/6 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

DTIC FILE COPY

(2)

SECURI AD-A181 408

DTIC ELECTE JUN 1 1 1987 S D

DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited. Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) KES.U.86.11	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 87-0791
6a. NAME OF PERFORMING ORGANIZATION Kestrel Institute 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION AFOSR
6c. ADDRESS (City, State and ZIP Code) 1801 Page Mill Road Palo Alto, CA 94304-1216	7b. ADDRESS (City, State and ZIP Code) Same as 8C
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR 8b. OFFICE SYMBOL (If applicable) NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F49620-85-C-0015
8c. ADDRESS (City, State and ZIP Code) Bldg 410 Bolling AFB, Washington, D.C. 20332	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. 61102F PROJECT NO. 2304 TASK NO. A2 WORK UNIT NO.

11. TITLE (Include Security Classification) Knowledge Based Synthesis of Efficient Structures for Concurrent Computation Using Fat-Trees and Pipelining

12. PERSONAL AUTHOR(S)
Richard M. King, Tom Brown

13a. TYPE OF REPORT Annual Technical	13b. TIME COVERED FROM ________ TO ________	14. DATE OF REPORT (Yr., Mo., Day) 31 December 1986	15. PAGE COUNT 65

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Fat-trees pipelining concurrency transformation synthesis parallel processing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In our previous work we developed techniques to synthesize lattice and tree parallel structures from first order logic specifications. We have now developed new techniques synthesize new structures.

First the new techniques enable the synthesis of trees in which the width of the interconnections and the power of the nodes increases as the distance from the leaves increases. This type of tree has been described in [Lei85] and given the name "fat-tree". The primary result of [Lei85] is that fat-trees are universal in that the performance of any network at all can be equaled by a fat-tree, to a constant and some factors logarithmic in the size of the structure to be simulated. The constant is immense, making fat-trees at present not a general method for simulating other strictures.

(please see reverse for continuation of abstract)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Capt Thomas	22b. TELEPHONE NUMBER (Include Area Code) 202/767-4954 22c. OFFICE SYMBOL NM

Block 19 (continued)

The idea of such a varying-width tree can, however, be used in specific cases as a synthesis target. We describe techniques for using extensions of our previous work [King85] to build specialized fat-trees to satisfy certain first order logic specifications. These fat-trees are efficient, unlike the general ones of [Lei85] because they are specialized.

The second extension is a proof that an appropriately defined parallel structure can be modified to produce a structure capable of pipelining, or processing different parts of several problem instances simultaneously in a manner similiar to an assembly line. The proof is a constructive one; a synthesis method based on the proof is feasible.

KES.U.86.11

AFOSR-TR- 87-0791

Kestrel Institute

Knowledge Based Synthesis of Efficient Structures for Concurrent Computation Using Fat-Trees and Pipelining

prepared by

Richard King
and
Tom Brown

Cordell Green,
Director

December, 1986

Prepared for:

Air Force Office of Scientific Research
Building 410
Bolling AFB, DC 20332-6448

Research sponsored by the Air Force Office of Scientific Research (AFSC) under contract F49620-85-C-0015. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notices hereon.

Kestrel Institute • 1801 Page Mill Rd. • Palo Alto, CA 94304-1216 • (415) 493-6871

87 6 10 227

Overview

We make the distinction between implicit concurrency and explicit concurrency. Implicit concurrency is that which is neither ruled out by nor required by the specification[1]. This paper is divided into two parts. The first part, chapters 1-7, describe the synthesis of parallel structure from first order logic specifications. We describe the explication of implicit concurrency. In the second part, we describe **KTL**, a wide-spectrum language which can be used to describe specifications with explicit concurrency, and show that it has a wide enough spectrum to be used as a synthesis tool on specifications with explicit concurrency.

In the first part, we briefly describe some of our previous work in making certain types of regular parallel structures, and describe some specialized languages that are well suited for describing such structures. We then describe an extension to this work that can enable the synthesis of specialized fat-trees, which is a type of parallel structure that is more expensive than either regular lattices of computing elements or ordinary balanced binary trees of elements, but which can handle an interesting class of problems of size n in $O(\log n)$ time, rather than the $O(n^c)$ for some strictly positive c time that would be required for either a lattice or an ordinary tree.

In the second part we explain why none of the existing types of specification systems is adequate to our needs, we give a formal definition of **KTL**, and we explain why it is a good description language for specifications with implicit concurrency.

[1]Some specifications rule out substantial concurrency because there is no way to make a parallel structure that satisfies the specification asymptotically faster than the best sequential solution. This is a deep property of the specification, as opposed to the property of whether there is explicit concurrency. As an analogy, the predicate "there is a win for white" is a deep property of a chess situation; the property "the white king is on A3" is a shallow property.

Chapter 1

Abstract

In a ~~our~~ previous work the authors ~~we~~ developed techniques to synthesize lattice and tree parallel structures from first order logic specifications. They ~~We~~ have now developed new techniques that synthesize new structures.

First the new techniques enable the synthesis of trees in which the width of the interconnections and the power of the nodes increases as the distance from the leaves increases. This type of tree has been ~~described in [Lei85], and~~ given the name *fat-tree*. ~~The primary result of [Lei85] is that~~ fat-trees are universal in that the performance of any network at all can be equaled by a fat-tree, to a constant and some factors logarithmic in the size of the structure to be simulated. The constant is immense, making fat-trees at present not a general method for simulating other strictures.

The idea of such a varying-width tree can, however, be used in specific cases as a synthesis target. The authors ~~We~~ describe techniques for using extensions of ~~our~~ previous work ~~[King85]~~ to build specialized fat-trees to satisfy certain first order logic specifications. These fat-trees are efficient, unlike the general ones of [Lei85], because they are specialized.

The second extension is a proof that an appropriately defined parallel structure can be modified to produce a structure capable of pipelining, or processing different parts of several problem instances simultaneously in a manner similar to an assembly line. The proof is a constructive one; a synthesis method based on the proof is feasible.

Chapter 2

Introduction

This and the seven chapters that follow are the first part of this paper, in which we describe techniques for the synthesis of parallel structure from specifications which have only implicit concurrency. We extend our previous work in two ways: we show that a third type of parallel structure is synthesizable using feasable techniques, and we demonstrate that a structure that *pipelines*, or simultaneously processes different parts of several different instances of one problem in an assembly-line manner, is simply derivable from a structure with explicit concurrency that does not pipeline.

2.1 Extension of Previous Work to Highly Interconnected Structures

There are three important classes of regular topologies described in the literature: tessellations; trees; and highly interconnected structures. In previous work we have described synthesis techniques to create both regular latticework structures and simple binary trees from first order logic specifications. The binary trees we were able to synthesize could have vectors as both the input and the output, provided that the computation resulting in a value at each node N depended only on other elements of the vector in a manner that could be summarized as a function of a fixed set of functions, from vectors to scalars, of substrings contiguous with N.

We are extending our previous work in two important ways. Our previous work was unable to synthesize highly interconnected structures such as butterfly networks and hypercubes. These structures are useful for a subclass of problems which take vectors into vectors, in which each output value depends on a small but fixed subset of the input values. A well known example of such a problem is Discrete Fourier Transform. Other problems for which this technique is useful arise from graph theory[1] e.g., connectedness problems.

A new set of techniques allows the synthesis of some of these highly interconnected structures. These techniques are based on the use of *closures* as a device to schedule communication, resulting from divide and conquer, between halves of a tree that are connected by a given node. An addition to this technique is that vectors that grow with the size of the subtrees can be communicated among internal nodes of the tree. The fact that these vectors are allowed to grow asymptotically allows information to be collected from two of the nodes connected to a given intermediate node and sent to the third. Nodes that are higher in a computation will be more powerful, both in computation and I/O capabilities, then those lower in the tree. These more complex nodes can be designed by recursive application of transformational techniques.

2.2 Use of Pipelines as a Synthesis Tool

Another extension of previous Kestrel work in concurrency synthesis results from a theorem concerning the ability of any parallel structure to pipeline.

Suppose all of the following:

1. There is a parallel structure that performs a computation that solves any instance of some problem[2].

[1]In some graph-theoretic problems, different communication takes place depending on whether a particular pair of nodes is or is not connected. This restricted variability in the data dependency can be handled by considering the computation to depend on all of the inputs that *could* be required, where this set is fixed.

[2]Note that the problem size would be fixed, since the structure is.

2. The number of messages sent from any processor to any other processor is not affected by the problem instance.

3. T_n is the maximum, for all wires leaving node n, of the number of transmissions on that wire.

4. C_n is the maximum computation time of node n for any problem instance.

Then the pipelining theorem shows that successive problem instances can be solved by a certain derived structure that is derived from the original structure and has the same topology, provided that the separation between successive problem instances exceeds $\max_{n \in structure} K_1 T_n + K_2 C_n$, where K_1 and K_2 are constants.

Applications of the pipelining theorem include synthesis of components for a larger structure, as well as direct application where the first-order logic specification really calls for a series of independent problems to be solved.

We will have therefore shown synthesis methods for all compactly describable topologies except for certain ones, such as the binary hypercube, that are appropriate for problems in which no predictions can be made about the pattern of communication. Structures for such problems could be "synthesized" by fixing on one standard topology for problems of a given size, since no information is available to make a choice among several such structures.

The organization of the rest of this paper is as follows: Chapter 3 gives a language to describe crystalline parallel processing structures, and Chapter 4 gives a similar structure for trees. Chapter 5 describes the synthesis of fat-tree structures, focusing on the synthesis of a parallel structure for the Discrete Fourier Transform. Chapter 6 shows that certain parallel structures can be transformed into pipelining structures, and Chapter 7 gives references for the first six sections. Chapter 8, which has its own references and introduction, describes KTL, a language for describing specifications in which the concurrency is part of the specification rather than the result of an optimization transfomation.

Chapter 3

Crystalline Structure Descriptions

We now give a language to describe crystalline structures. In crystalline concurrency, processors are connected together in such a manner that the processors can be assigned Cartesian coordinates and the pattern of connections of one processor with its neighbors looks similar to those of other processors. More precisely, there is some n such that each processor can be assigned an index in Z^n, and each processor is connected to those neighbors whose coordinates differ from its own by one of a set of constant offset vectors (when such a processor exists). The description of a topology can include parameters in such a way that a family of similar topologies, differing only in size, can be described by instantiating those parameters to different constants.

This definition has several consequences: from any processor within a fixed crystalline structure, the number of other processors reachable by paths of length m from any node is $O(m^c)$ for some constant c (usually a small positive integer); the bisection width[1] is also polynomial in the size of the crystal; and the diameter[2] is also polynomial in the size of the structure.

[1]The *bisection width* of a graph is the minimum, over all divisions of the graph's nodes into two groups of approximately equal size, of the number of edges between the two halves.

[2]The *diameter* of a graph is the maximum, over all pairs of nodes, of the distance between the nodes. The *distance* between two nodes is the minimum, over all paths from one node to the other, of the path length.

3.1 Assumed Crystal Description Language

We define a notion of *families* of processors. A family of processors has a name and a dimension; it is defined as a mapping taking Cartesian products from elements of index sets (usually integers) to processors. Defining a family of processors involves the specification of quantification of indices.

Interconnections, responsibilities (assertions that a given processor is responsible for computing a certain value) and per-processor procedure declarations must also be provided to define a parallel structure.

The prototyping language we are using for this project is lexically scoped. Because of this and the fact that interconnections between two families of processors can be mutual, the arrays of processors must be declared in a scope that strictly encloses the declarations of interconnections and responsibilities. (In previous designs, processor names were effectively of global scope.)

An advantage of having lexical scoping is the possibility of having a hierarchy of processors within processors, so the computation specified for one processor can be performed by multiple processors contained within.

3.2 The Topology Language

The first element of the syntax is the declaration of a processor family:

```
(let (P : (mapping (pair int, int) → processor))
     .  .  .
     )
```

This declares that there exists a two-dimensional family of processors with a family name of P. Because it says nothing about the range of values that the indices can take, and the idea of bounding sets of values occurs frequently, we provide a construct for bounding sets of values as follows:

$$(\langle \textbf{decl} \rangle\ (i,\ j)\ [\textbf{suchthat}\ i \in [1 \ldots n] \land j \in [1 \ldots i]] \Rightarrow\ \ldots)$$

Here ⟨**decl**⟩ can be any of five operators; which one is selected determines what is being declared by the bounding construct. The variables of the declaration (here i and j) have as their scope the entire declaration. The ⇒ is best thought of a compile-time logical implication.

The first possibility for ⟨**decl**⟩ describes the shape of a family of processors:

$$(\textbf{shape}\ (i,\ j)\ [\textbf{suchthat}\ i \in [1 \ldots n] \land j \in [1 \ldots i]] \Rightarrow [P_{i,j} \Downarrow])$$

or more generally

$$(\textbf{shape}\ (\langle \text{shapevar} \rangle \text{s})\ [\textbf{suchthat}\ \langle \text{shapebound} \rangle] \Rightarrow [\langle \text{processor expression} \rangle \Downarrow])$$

where ⟨shapevar⟩s is the bound variables list of the construct, ⟨shapebound⟩ is a predicate that is true exactly when the quantified variables of ⟨shapevar⟩s take values that make ⟨processor expression⟩ defined. ⟨processor expression⟩ should have some of the variables of ⟨shapevar⟩s free. The notation ⟨processor expression⟩ ⇓ can be read "⟨processor expression⟩ exists" (loosely borrowed from the notations of recursive function theory).

The next construct, the **responsibilities** construct, describes the set of values that a given processor is responsible for. Example:

$$\begin{aligned}&(\textbf{responsibilities}\ (\langle \text{shapevar} \rangle \text{s})\ [\textbf{suchthat}\ \langle \text{shapebound} \rangle] \\ &\qquad \Rightarrow [\langle \text{processor expression} \rangle\ \textbf{has}\ \langle \text{value designation} \rangle])\end{aligned}$$

The concept here is that for all acceptable vectors of values of ⟨shapebound⟩ the processor named by ⟨processor expression⟩ is responsible for computing the values described by

⟨value designation⟩. It is also the source of ("**has**") these values. A ⟨value designation⟩ will usually be a reference to one element of a vector or an array. The ⟨value designation⟩ and ⟨processor expression⟩ must be such that different instantiations of the ⟨shapevar⟩s will not yield the same value designation for different processors.

A third construct describes communication among processors, together with the reason for that communication. Example:

(**communication-network** (**with** (⟨shapevar⟩s) [⟨shapebound⟩] (⟨linkage⟩s)))

where (⟨shapevar⟩s) and ⟨shapebound⟩ are as before, and ⟨linkage⟩s are communications, described as follows:

(⟨proc expr1⟩ **type** ⟨proc expr2⟩ [→ ⟨proc expr3⟩] (⟨motivation⟩s))

Here the **type** is one of **hears**, **talks-to** or **links**, declaring respectively: the existence of a "wire" *to* the processor described by ⟨proc expr1⟩ from the one described by ⟨proc expr2⟩; a wire *from* ⟨proc expr1⟩ to ⟨proc expr2⟩; and a relayed wire from ⟨proc expr2⟩ to ⟨proc expr3⟩ via ⟨proc expr1⟩. Note that each of the **talks-to** and **hears** information can be inferred from the other, and that the **links** information can be derived from either of these.

The ⟨motivation⟩s are descriptions of sets of values that are expected to flow over the link being declared. Here is the "generic motivation":

(**m-type** (⟨shapevar⟩s) [⟨shapebound⟩] (⟨referee⟩s))

where ⟨shapevar⟩s and ⟨shapebound⟩s are as before, and ⟨referee⟩s are ⟨value designation⟩s of values that are to be carried by the declared wire.

3.2.0.1 The Processor Description Language

It is also necessary to describe the work done by each processor.

Under the assumptions we are using, there is no need to explicitly specify acts of communication. If there are two processors P_1 and P_2 that are related by a communication clause, one of whose motivations is value v, then the value identified by v is communicated from P_1 to P_2 as soon as P_1 comes to know it. The need to communicate will not change the asymptotic complexity of the processors unless each processor is connected to a nonconstant number of other processors, a situation we expect to avoid.

Chapter 4

The Tree Description Language

Families of processors can be tree shaped as well as crystalline. Because of decidability problems that would otherwise arise (see [Kin85]), we provide no provision to describe the balance of the tree. For our purposes a tree must be balanced[1]. In order to allow the formation of the fat-trees of [Lei85], some limited information must be available to describe the positions of internal tree nodes.

Because a processor tree is so stereotyped, the type constructor

```
(let (T : processor - tree)
   . . .
   )
```

suffices to describe those attributes of T that are not either described implicitly in communications constructs to be described below, or an option of the implementation.

The declaration of the family T actually causes there to come to be three families: T.**root**, T.**internal**, and T.**leaves**. T.**internal** is indexed by subsequences of the sequence of leaves, although it is not true that T.**internal**$_{\vec{s}}$ exists for all subsequences $\vec{s}$ of leaf indices. Each intermediate node knows its own index and those of its children.

[1]An unbalanced tree can be described by specifying a connection between the root of some subtrees and chosen leaves of the supertree. Because the description is fixed, only one tree shape can be used of a given size.

In the succeeding subsections we will describe the constructs for specifying the communication within the tree and the information available to the outside world through the root. It is also possible to arrange for communication among corresponding elements of distinct trees by providing communications clauses specifying use of a value, in intermediate nodes of one tree, that are had by intermediate nodes of another.

4.1 Shape Restrictions

The first element of the syntax is the declaration of a processor family:

(let (T : **processor-tree)**
. . .
)

This declares that there exists a tree of processors with family names of T.**left**, etc. It says nothing about the number of leaves or the nature of the leaves' index set.

The shape of the leaf set is determined thus:

(shape (i) [**as** $i \in [1 \ldots n] \wedge P(i)] \Rightarrow [T.\mathbf{leaf}_i \Downarrow]$)

o: more generally

(tree-shape ($\langle$shapevar$\rangle$s) [**as** $\langle$shapebound$\rangle$] $\Rightarrow$ [($\langle$processor expression$\rangle$ $\Downarrow$)]

where here $\langle$shapebound$\rangle$s must be of a form acceptable to a "sequence former" (the basic criterion is that it generates an *ordered* stream of values rather than an unordered set of values). $\langle$shapevar$\rangle$s is the bound variables list of the construct, $\langle$shapebound$\rangle$ is a predicate that is true exactly when the (quantified) variables of $\langle$shapevar$\rangle$s take values that make $\langle$processor expression$\rangle$ defined. Processors of $\langle$processor expression$\rangle$ must be indexed references to T.**leaf**, where T is of type $processor - tree$. $\langle$processor expression$\rangle$ should have some of the variables of $\langle$shapevar$\rangle$s free.

The next construct, the **tree-responsibilities** construct, describes the set of values that a given processor is responsible for. One example of such, in this case for leaves, follows:

> (**tree-responsibilities** (⟨shapevar⟩s) [**suchthat** ⟨shapebound⟩] ⇒
> [⟨leaf processor expression⟩ **has** ⟨value designation⟩])

Here we say that for each instantiation of the variables of ⟨shapevar⟩ that satisfy ⟨shapebound⟩ the processor named by ⟨processor expression⟩ is responsible for computing, and is the source of, (**has**) the values described by ⟨value designation⟩. A ⟨value designation⟩ will usually be a reference to one element of a vector or an array. It must never be true that for two instantiations of ⟨shapevar⟩s for which the ⟨value designation⟩s are equal but the ⟨processor expression⟩s are not.

Note that an ordering is defined on the leaves by the order of the stream generated by the **suchthat** clause.

The leaves of a tree are a crystalline structure, and are defined similarly to any other crystalline structure. The root is also a processor whose communication can be described by crystalline structure communication constructs with a null vector of indices.

The differences arise when intermediate nodes are involved.

For the intermediate node $T.\mathbf{internal}_{\vec{i}}$ it is assumed that $\vec{i} = (\mathbf{concat}\ \vec{j}, \vec{k})$, where $\vec{j}$ and $\vec{k}$ are the subscripts of the children. No other information can be supplied for the way the sequence $\vec{i}$ is split between $\vec{j}$ and $\vec{k}$, as this is an option reserved for the system. The intermediate node $T.\mathbf{internal}_{\vec{i}}$ is described as **hearing** $T.\mathbf{internal}_{\vec{j}}$ and $T.\mathbf{internal}_{\vec{k}}$. If $T.\mathbf{internal}_{\vec{i}}$ **has** $V_{\vec{i}}$ it may **use** $V_{\vec{j}}$ and $V_{\vec{k}}$ as well.

Chapter 5

Fat-Trees as a Synthesis Target

Two most important properties of a parallel structure are its cost and its performance. These are, in turn, related to the *minimum area* required to wire a parallel structure containing a given number of processors; the bisection width, and the diameter.

The bisection width is very closely related to the minimum area required to implement a structure; the normal proof technique used to establish area/time tradeoff theorems is to show a lower bound on the bisection width of a topology that can solve the given problem in a given amount of time, and the normal proof technique used to establish lower bounds on the area of an implementation of a topology is to describe its bisection width. This has been a normal technique in this field for some time; see, for example, [**BrK79**].

Architectures occupying prominent places in the literature and along this continuum from high performance and high cost to low performance and low cost include tree structures and crystalline structures which have been the subjects of our previous work, and highly interconnected structures such as the cubically connected cycles, the binary hypercube, the perfect shuffle, and the butterfly.

5.1 Motivation for High Interconnection

In the ideal multiprocessor system, every processor would have access to all information in the system in unit time. This would involve every processor being able to directly communicate with every other processor. The expense of such a system would be so extreme, except for a very small number of processors; we have not, and will not, consider this possibility in this Report.

At the next level of desirability is for every processor to have access to information in an amount of time logarithmic in the size of the structure. In order for this to be possible, each processor must be the root of a tree that differs only by a constant from being balanced. These trees can overlap, which they must do to avoid a quadratic number of elements for the entire structure. This overlap causes some problems in handling communications patterns that cannot be predicted in advance, but the details of these problems need not concern us here, except to say that ways, including use of random routing, have been developed to deal with the problem. See, for example, [**ReV82**].

5.2 Restricted Highly Interconnected Structures

The synthesis of a highly interconnected structure only becomes interesting if there is some information available as to what communication patterns hold. If this is not the case, than there is nothing in the communications pattern to distinguish two networks of a given size, so any two structures with the same number of nodes would have the same optimal topology.

In interesting structures, however, there are restrictions on the communications patterns. A trivial example of this would be that if one processor were the source (resp. target) of all of the communications, then a tree could be used.

A less trivial example can be offered. Consider the Discrete Fourier Transform (DFT).

In what follows we will use the convention that $\vec{U}$ is a vector- or array-valued object, and that u_i is the notation for element extraction. If the vector is thought of as a mapping, u_i could have been written $(\vec{U}\ i)$.

The assertion that $\vec{W}$ is the DFT of a vector $\vec{V}$ can be specified as

$$\exists\vec{X}[\forall i[x_{-1,i} = v_i \wedge \forall j[x_{j,i} = G(j, x_{j-1,i}, x_{j-1,i\otimes 2^j})] \wedge w_i = x_{jmax,i}]]$$

where $\vec{X}$ is an array, i and j are indices into vectors and arrays, and $G(j, y, z) = y + \omega^{2^j}z$ and ω is the primary root of $\omega^{2^n} = 1$. $\otimes$ is the "exclusive or" operator. There are several ways of thinking about this specification and assigning processors to values, each of which produces a different, plausible, and interesting parallel structure.

If we assign a processor for each element of $\vec{X}$ and provide every wire implicit in the data communication we get the standard butterfly network for DFT, in which (for 2^n points) there are $n + 1$ banks of 2^n processors each, indexed by two numbers ranging from 0 to n and from 0 to $2^n - 1$, such that processor $P_{j-1,i}$ sends information to processor $P_{j,i}$ and to $P_{j,i\otimes 2^{j-1}}$.

If we assign a processor for sets in $\{\{x_{j,i} : -1 \leq j \leq n-1\} : 0 \leq i \leq 2^n - 1\}$, and make the interconnections implicit in the data flow, we get the binary hypercube of order n.

A third choice is to use closures and divide-and-conquer, but to relax the requirement that an argument to a closure be a scalar. This leads naturally to fat-trees [Lei85], in which a structure is a binary tree but higher nodes in the tree are more complex.

In this case, the intermediate nodes of this tree receive a closure and a value from the next lower level.

The general pattern is that the leaves provide a closure and a value. The closure accepts one value and performs one step of the DFT, and the value is sent upwards in the tree to be used as an argument for similar closures. The leaf executes the following two assignments (recall that assignment to a variable that another processor uses provides for communication):

$$C_{i,j} \leftarrow \lambda_z^{x_{j,i},x_{j-1,i}}[x_{j,i} \leftarrow G(j, x_{j-1,i}, z)]$$
$$v_{i,j} \leftarrow \; x_{j-1,i}$$

where $\lambda_{\vec{X}}^{\vec{Y}}$[body] is a notation for a *closure generating form.* A closure generating form is one that evaluates to a closure, and this example form would create a closure that accepts as its arguments the variables of $\vec{X}$, and in which the current values of the variables of $\vec{Y}$ are embedded. Application of the closure will involve evaluating body in an environment where $\vec{X}$ has been bound to the arguments and the variables of $\vec{Y}$ are bound as they were when the closure generating form was evaluated.

An intermediate node of the tree must be able to either switch information from corresponding nodes in one of its subtrees to the other, or to send information from each subtree to its parent. Working through a synthesis by closure (as in [Kin85]), but not maintaining a restriction that vector concatenation not be used to create the values that are passed up the tree, we find that the program in an intermediate node at level k (the bottom level is level 0) is

```
⟨Program 1⟩
$C.\mathbf{up}_{i,j} \leftarrow \mathbf{if}\ j = k$
        $\mathbf{then}\ \lambda_{\vec{Z}}^{C.\mathbf{left}_{i,j},C.\mathbf{right}_{i,j},\vec{Z}.\mathbf{left},\vec{Z}.\mathbf{right}}[C.\mathbf{left}(\vec{Z}.\mathbf{right}) || C.\mathbf{right}(\vec{Z}.\mathbf{left})]$
        $\mathbf{else}\ \lambda_{\vec{Z}}^{C.\mathbf{left}_{i,j},C.\mathbf{right}_{i,j},\vec{Z}.\mathbf{left},\vec{Z}.\mathbf{right}}[C.\mathbf{left}(\vec{Z}.\mathbf{left}) || C.\mathbf{right}(\vec{Z}.\mathbf{right})]$

$\vec{Z}.\mathbf{up} \leftarrow \mathbf{concatenate}(\vec{Z}.\mathbf{left}, \vec{Z}.\mathbf{right})$
```

In this structure the vectors from the descendants are collected and sent to the parent, and also at the appropriate level of the tree (which varies from cycle to cycle) the vectors are interchanged for the downward call[1].

The structure for this looks like

```
(let ($DFTP$ : processor-tree)
    (tree-shape ($i$) [as $0 \leq i \leq 2^n - 1$] $\Rightarrow$ $DFTP$.leaf$_i$ $\Downarrow$)
    (tree-responsibilities ($i, j$) [suchthat $0 \leq i \leq 2^n - 1 \wedge -1 \leq j \leq n$]
            $\Rightarrow$ $DFTP$.leaf$_i$ has $x_{j,i}$)
    (tree-responsibilities ($j$) [suchthat $-1 \leq j \leq n$]
            $\Rightarrow$ $DFTP$.internal has $C_j, z_j$)
    (communications () [suchthat true]
            $\Rightarrow$ $DFTP$.internal uses $C_j, z_j$)
    (in $DFTP$.internal$_{\text{level } k,\ 0 \leq k \leq n-1}$) :
          (Program 1))
```

5.3 Additional Synthesis Steps

Firstly, other techniques from our armory must be used to synthesize the internal structure of a node. In this example, the procedural part of the closures has internal structure, which can result in the synthesis of tree nodes with corresponding internal structure.

Consider Program 1 again. The computations inside the internal nodes are, themselves, operations on vectors, and can therefore be handled by the methods of [King85] to yield multiprocessor internal nodes and multichannel connections within the tree, as follows:

```
(let ($DFTPI$ : processors,
      $DFTPL$ : processors,
      $DFTPR$ : processors,
```

[1]Notice that the argument passed to the closures at or above the critical level are irrelevant, since at the critical level the "other" vector is used instead of the argument.

```
    $DFTPU$ : processors)
  (shape ($m$) [as $0 \leq m \leq 2^{k+1}$] $\Rightarrow DFTPI_m \Downarrow$)
  (shape () [] $\Rightarrow DFTPL \Downarrow$)
  (shape () [] $\Rightarrow DFTPR \Downarrow$)  % *interface processors*
  (shape () [] $\Rightarrow DFTPU \Downarrow$)
  (responsibilities ($m$) [suchthat $0 \leq m \leq 2^k$] $\Rightarrow DFTPI_m$ has $C_m, z_m$)
  (communications ($m$)
       [suchthat $0 \leq m \leq 2^k$] $DFTPI_m$ hears $DFTPL$ (uses $z$.left$_m$)
                                          $\wedge\, DFTPI_m + 2^k$ hears $DFTPR$ (uses $z$.right$_m$)
                                          $\wedge\, DFTPI_m$ hears $DFTPI_m + 2^k$ (uses $z$.right$_m$)
                                          $\wedge\, DFTPI_m + 2^k$ hears $DFTPI_m$ (uses $z$.left$_m$)
                                          $\wedge\, DFTPU$ hears $DFTPI_m$ (uses $z$.up$_m$)
                                          $\wedge\, DFTPU$ hears $DFTPI_m + 2^k$ (uses $z$.up$_m + 2^k$))
  (in $DFTPL$):
    for $0 \leq i \leq n$ do
      $zr = [z_m : 0 \leq m \leq 2^k - 1]$;
      $C \leftarrow \lambda_{zu}^{Cl_i=Cl_i, Cr_i=Cr_i, zl_i=zl_i, zr_i=zr_i}$
           [if $i = k$ then $(Cl_i(zr_i) || Cr_i(zl_i))$ else $(Cl_i(zl_i) || Cr_i(zr_i))$
      % $zl$, etc. is an abbreviation for $z$.left, etc.
    end
```

Here what happened is that a crystalline synthesis was performed *within* the internal nodes of the tree.

Second, and more important, the requirement that the auxiliary values computed and passed through the tree be scalars is removed. This requires the ability to have as a parameter the level of a node in the tree, because the size of the extended objects in general depends on the level in the tree (and therefore the number of leaf nodes in the subtree).

5.4 Applications

The Discrete Fourier Transform is one example of a genre of mappings from vectors to vectors, in which each element of the output depends on each element of the input, but in a regular enough manner that a cumulative partial information vector can be computed and used. Other examples can be drawn from the literature of divide-and-conquer. (e.g., graph theory problems)

Naive divide-and-conquer can only be used to synthesize a tree structure when the output is a scalar ["census functions"]. In previous work at Kestrel [**King84**] the application of closures to synthesize trees for divide-and-conquer has been shown to be able to overcome the problem of excessive communication through the root in cases in which the input and output vectors are the same size, and the output vector is pointwise dependent on the input vector and information summarizable in scalars from the contiguous subtrees. The fat-tree synthesis will enable most computations in which the communication pattern does not depend on the data to be synthesized into a specialized fat-tree, making an expensive but fast network.

Chapter 6

Fundamental Pipeline Theorem

In this Chapter we will consider functions from vectors to vectors.

Definition 6.0.1 *A* parallel structure *[PS] is a collection of computing* elements*;* internal *wires between pairs of elements,* input *wires from sources of data to elements, and* output *wires from elements to destinations of data; and programs loaded into the elements. A PS* solves *a family of problems P if, with proper initialization of the elements and delivery of the problem's input values to the input wires, the solution comes to be delivered to the PS's output wires. The* size *of PS is the number of elements it contains.*

In complexity theory, a Turing Machine computation is termed "oblivious" if the motion of the head of the Turing Machine does not depend on the problem instance. By analogy, we will define:

Definition 6.0.2 *A computation takes place in an* oblivious *manner if the multiset of origins and of destinations of the signals received by every element does not depend on the problem instance.*

In what follows we will assume that the duration of a transmission does not depend on its contents. We are therefore considering the length of the datum to be irrelevant. This tends to not be exactly true for simple descriptions of problems, but can be made true by restricting the domain of applicability of the structure. As an example, there is no parallel structure that solves the problem "add two integers" in an oblivious manner because for any fixed transmission duration and technology it is possible to provide inputs that will make the transmission take longer than the alloted time. The more specific problem, "add two numbers, both of whose magnitudes are $\leq 2^{31}$", can be so solved.

This tendency for transmission lengths to grow as problem sizes increase is a serious matter in asymptotic behavior analysis, because larger versions of families of related problems tend to have intermediate values that take longer to transmit. We do not think this to be a serious problem here, however, for two reasons: increasing the number of problems in a pipelining system at any given time, or the total number of problems fed through such a system, does not tend to increase transmission times for intermediate values, and (as the integer addition example shows) the limitations imposed by a constant-transmission-size requirement tend to translate into argument size limits. If we were proving results about the behavior of *families* of problems and corresponding families of parallel structures, this would be more of a problem because some such families have intermediate structure sizes that grow asymptotically with the size of the problem.

In this model an element contains two parts:

- a processing part that contains a state, and a program that maps a pair of
  - vectors of input values (some of which are null because they correspond to wires that delivered no signal), and
  - internal processing states including an end state

  into a pair of processing states and sets of pairs of messages and output lines
- a queueing/output unit

with the message/output line pairs being sent to the queueing/output unit to be sent on the output wires in the order received. We assume that all computations are done in a timing-independent manner, so that if an element "expects" to see two signals on each of two lines, arrival of the signals in either order, or simultaneously, will have the same effect. Since we assume oblivious computations, we assign unique names to the transmissions that take place within PS as it solves an instance of the problem.

We assume in our model that the reception is no slower than the transmission – ie., there is never any inhibition of transmission; as the time required to receive a signal is assumed to be subsumed in the transmission time.

Definition 6.0.3 *The* pipelining structure *for a PS [*P(PS)*] is the structure that results from PS by replacing each communication path whose messages are of type T by one whose type is* integer $\times$ T*, the state S in each element is replaced by a mapping* M *from integers to states, and each program F is replaced by a program that does the following thing for each index seen on an input line.*

For each i such that $\langle i,x\rangle$ occurs in the input vector, a new vector is formed by replacing instances of $\langle i,x\rangle$ with merely x, and instances of $\langle j,x\rangle : j \neq i$ with Λ, where Λ is the void value. The new program computes $\langle M(i),\vec{O}\rangle \leftarrow F(M(i),\vec{V})$ (except that if $M(i)$ is not defined the starting state is used instead and if F produces the end state than $M(i)$ is left undefined) and sends $[(\text{if } x = \Lambda \text{ then } \Lambda \text{ else } \langle i,x\rangle) : x \in \vec{O}]$.

The queueing/output units in the elements of $P(PS)$ will always resolve conflicts by sending an $\langle i,x\rangle$ pair with the lowest possible i. (This can happen because F can "decide" to send several transmissions, from several input vectors and internal states from the mapping, on a single output wire.)

Intiutively, there are several sets of data coursing through the PS, and it is necessary to keep the various data and internal states straight to ensure that computations are performed on corresponding input values and internal states. The program in an element is replaced by a second program that acts as follows:

We will now show that $P(PS)$ can solve multiple problem instances, provided that they are sufficiently separated in time.

First we get some preliminaries out of the way:

Basic Observation 6.0.1 *If PS solves a problem instance PI, then P(PS) will also solve such a problem instance, provided we pair each piece of the input with a (unique) integer i and anticipate output values being delivered with that same integer. A problem instance that has been so modified is called $P(PI, i)$.*

This observation follows simply from the fact that for every value v received by an element of PS the corresponding element of P(PS) will receive $\langle i, v\rangle$, the corresponding state transition will take place in $M(i)$, and therefore if the element of PS would have transmitted w at some time, the element of P(PS) will transmit $\langle i, w\rangle$.

Definition 6.0.4 *The* content *of a pipelining structure is the size of the domain of that mapping within elements of the structure for which that domain is largest. Recall from the definition of a pipelining structure that i is removed from $M(i)$ when the latter comes to be an ending state.*

Definition 6.0.5 *The* separation *of a stream of problem instances is the amount of time between the start of transmission of successive instances. It is assumed that there is no overlap, i.e., that a problem instance will be delivered in its entirety before a successor is begun.*

Definition 6.0.6 *The* duty *of an element [$D(E)$] is the number of transmissions it issues on any single line during one problem instance. In our model, this is proportional to* *the amount of time the element is in use; we make this proportionality constant* 1 *by appropriate choice of units. The duty of a PS is* $\max_{E \in \mathrm{PS}}[D(E)]$.

First we established a lower bound to the separation of a stream of problem instances:

Basic Observation 6.0.2 *For no PS with duty D is it possible to solve n problem instances in fewer than nD time units.*

Let the dwell be W, and the separation be S. After T time units, $\frac{T}{S}$ problem instances have been started, and all but $\frac{W}{S}$ of them have exited the system. For this to be true, the element whose duty is D must have delivered $\frac{D(T-W)}{S}$ messages to the successor with which its duty is D. Since $S < D$, $\exists T[D\frac{T-W}{S} > T]$. After T time, the element whose duty is D is required to have handled $\frac{T-W}{S}$ problem instances, and therefore to have made $D\frac{T-W}{S}$ transmissions on one of its lines; this is imposible, as $D\frac{T-W}{S} > T$. ∎

Definition 6.0.7 *A* valid itinerary *of a PS is a map:I* : integer → (multiset element) *such that if a problem instance is delivered at time 0 then exactly those elements in* $I(i)$ *contain sufficient information to transmit messages at time* i*, assuming that all elements mentioned earlier in the itinerary did in fact transmit at that time. In what follows we will merely call this an itinerary. In cases where there will be no confusion, we will draw a valid itinerary as a sequence.*

Definition 6.0.8 *The* dwell *of a pipelined structure is the maximum, over an indefinite stream of problem instances, of the time interval between the entry of the first part of the instance into the structure and the exit of the last part.*

Basic Observation 6.0.3 $\sum_i[\#I(i)] \leq \mathrm{size(PS)} \cdot \mathrm{duty(PS)}$, *immediately from the definitions.*

Definition 6.0.9 $\sum_i[\#I(i+1)]$ *is the* activity *of the PS.*

Basic Observation 6.0.4 $I(i) \neq \emptyset \Rightarrow 0 < i \leq \text{activity(PS)}$, *because* $I(i) = \emptyset \Rightarrow I(i+1) = \emptyset$*, since in our model every transmission is stimulated by a reception.*

We now show the main result of this Chapter. The idea of this proof is that, given a valid itinerary for the unpipelined structure, we can construct a valid itinerary for the pipelined version of that structure that has certain desirable properties when fed a continuous stream of problem instances.

The properties are:

- the separation of a stream of problem instances need not exceed D, where D is the duty of the nonpipelined structure;
- the time from start to finish of one problem instance is no greater than the product of the activity and D;
- no internal element queue need accomodate more than D values;
- there will be no more than activity problem instances in the pipelined structure at one time.

The basic idea of the proof is to construct, from the valid itinerary for the non-pipelined structure, a different itinerary such that any number of copies of the second itinerary can be superimposed on each other without interfering with each other. I construct this by taking the first itinerary $\langle i_0, i_1, i_2, \ldots \rangle$ and adding spaces to create $\langle i_0, \Lambda, \Lambda, \ldots, i_1, \Lambda, \Lambda, \ldots, i_2, \Lambda, \Lambda, \ldots, \ldots \rangle$, in which the separation of the non-null portions of the itinerary is the duty.

If an arbitrary number of these is then superimposed with the non-null portions overlapping, we have, during certain cycles, one instance each of i_0, i_1, etc. Since by definition the duty of no element is greater than the duty of the structure, it will be possible for each element to bleed its queue during the quiet spaces.

Theorem 6.0.1 *Consider an indefinite stream of instances $PPI_1 = P(PI_1, 1), PPI_2 = P(PI_2, 2), \ldots$. Suppose that the separation of problem instances equals or exceeds one more than the duty of $P(PS)$. Then there will be no point at which internal queues in the element grow without limit, and the content of the PS is bounded.*

Proof: We call the activity of the structure A and the duty D. Create I' as $I'((D+1)i) = I(i)$, $j \bmod D+1 \neq 0 \Rightarrow I'(j) = \emptyset$.

Let $F = \max_i[i < (D+1)A \wedge i \bmod (D+1) = 0]$. Using $\uplus$ to denote multiset "union", we consider the mapping $M(j) = \biguplus_{i \in \{j, j+D+1, j+2(D+1), \ldots, F\}} I'(i)$. Now if we define $M(j,k) \equiv \biguplus_{j \leq i < k} M(j)$, then $M(j, j+D+1)$ includes at most D instances of any single element. The reason for this is that the range of I included each element at most $D+1$ times, and the operations used to compute $M(i)$ merely fold the range of I so $M(i, i+D)$ will produce the same multiset.

The folding of the itinerary corresponds to the propagation of multiple problem instances through $P(PS)$. It will necessarily possible for an element to transmit, in the D cycles following cycle i, those values made necessary by receptions during cycle i or previously. Each value transmitted during one of these D cycles will be received when or before it is scheduled to be received, which is at cycle $i + D + 1$.

Inserting problem instance i at the start of cycle $(D+1)i$, we find that problem instance i will depart the system (all elements will have $M(i)$ undefined) before cycle $i + (D+1)A$.

∎

There are two ways this theorem could fail to be "tight": the separation of $D+1$ could be unnecessarily generous, it might be provably possible to have a content smaller than A, and it might be provably possible for the dwell to be less than DA.

The possibility that the separation is less than D has been dealt with above.

We suspect, however, that a dwell time comparable to A is possible. The general idea is that once each internal element queue has as many as A elements, at least one transmission from each problem instance in the system will take place each cycle. One difficulty in proving this is the possibility that the size of the elements' queues oscillates wildly.

Chapter 7

References

[BrK79] R. Brent and H. Kung, "The Area-Time Complexity of Binary Multiplication" *CMU Tech Report CMU-CS-79-136*, July 1979

[BHR84] S. Brookes, C. Hoare and A Roscoe, "A Theory of Communicating Sequential Processes" *Journal of the ACM v. 31 # 3*, July, 1984, pp. 560–599

[Bro86] Thomas C. Brown, "Symbolic Computation Domains: A Reflective λ-Calculus and its Type Theory" *Kestrel Tech Report # KES.U.86.2*, March, 1986

[EmCl] Emerson, E. Allen, and Clarke, Edmund C., "Using branching time temporal logic to synthesize synchronization skeletons" *Science of Computer Programming 2*, 1982, pp. 241–266

[Gel85] Gelernter, David, "Generative communication in Linda" *ACM TOPLAS 7(1)*, Jan. 1985, pp. 80–112

[Hoa78] C. A. R. Hoare, "Communicating Sequential Processes" *Communications of the ACM Vol. 21 # 8*, August 1978, pp. 666–677

[Kin85] R. King, "Knowledge-Based Transformational Synthesis of Efficient Structures for Concurrent Computation" *Rutgers Ph. D. Thesis*, May 1985

[Ken81] K. Kennedy, "Program Flow Analysis, Theory and Allpications; A Survey of Data Flow Analysis Techniques", *Prentice Hall 1981, Chapter 1*

[Lei85] C. Leiserson, "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing"*IEEE Transactions on Computers, C-34 #10*, October 1985, pp. 892–901

[Mos83] J. Mostow, "Program transformations for VLSI"*8th International Joint Conference on Artificial Intellegence*, Aug 1983, pp. 40–43

[ReV82] J. Reif and L. Valiant, "A Logarithmic Time Sort for Linear Size Networks", *Harvard Tech Report #TR-13-82*, 1982

[sb81] Stefik, Mark and Dan Bobrow, "Linked Module Abstraction: a methodology for designing the architectures of digital systems"*Memo KB-VLSI-81-9, Xerox PARC*, June 1981

[sl85] Shahdad, Moe, Roger Lipsett, Erich Marschner, Kellye Sheehan, Howard Cohen, Ron Waxman, and Dave Ackley, "VHSIC Hardware Description Language"*IEEE Computer 18 # 2*, February 1985, pp. 94–104

[Smi83] D. Smith, "Derived Preconditions and Their Use in Program Synthesis" *Tech Report, Naval Postgraduate School, Montery, CA 93940*, November 1983

Chapter 8

A Specification Basis for Computing-System Synthesis

Contents

8.1 Overview

This chapter describes preliminary work on a unified basis for specifying *problems, algorithms, parallel computing architectures and systems.* Work described in previous chapters introduces and develops sublanguages for specifying these entities on an *ad hoc* basis. For example, in addition to the *predicate language* for specifying problems there are the *topology language* for specifying parallel architectures and the *processor description language*, an imperative language for specifying algorithms to be implemented or executed by processors.

Ultimately it will be useful (if not essential) to describe and relate all of the specifications mentioned above in a single language. The rationale for this unification is simply that problem specifications and related constraints go into the parallel computing system synthesizer, parallel computing system specifications come out, and validity of the transformational synthesis methodology depends on the inputs and outputs being related by a sequence of correctness-preserving specification-transformations. While there is a good argument for separate languages in widely differing realms (e.g., system specifications versus realizations in a specific VLSI device technology), we argue that the specification realms considered above are sufficiently interdependent (and intermixed during various stages of synthesis) to require unification in a single language wherein correctness of transformations has a precise semantic definition.

Definitions. To see how the unification may be accomplished, it is necessary to review briefly what we mean by the terms. A *problem* specification describes some input-out relation or environment-interaction to be realized, and optionally describes constraints on realizations (such as solution on a specified architecture within specified real-time constraints). An *algorithm* specification describes an effective computation based on given primitives. An *architecture* is an abstract object-type, whose instances are computing systems. By a *parallel computing system* we mean a programmed digital system with multiple processing elements that operate simultaneously. As usual, a *program* consists of algorithm and input data (perhaps encoded for execution by an architecture).

A transformational approach to synthesis of parallel computing systems from restricted problem specifications is developed in [King 85] and extended in previous chapters. This work has focussed on a restricted but important class of problems and solutions (single-assignment algorithms), though it appears to be generalizable. A goal of the work reported below has been to enlarge the domain of problems and architectures that can be described, and to provide an effective semantic foundation for synthesis of parallel computing systems by correctness-preserving transformations.

8.1.1 Problem Statement

Soon we will have a VLSI technology that can provide adequate computing power for most of the critical real-time problem-solving tasks that confront us. But harnessing this power poses new challenges for computing system architects and software engineers. There is a growing consensus that *we will be unable to develop either the architectures or the controlling software in a timely, reliable and economical manner without major innovations in the technology of system design, development and maintenance* .

One problem is that, while a few experts can show what is possible by honing algorithms to solve key problems brilliantly on selected parallel architectures[HG 86], it is unlikely that large quantities of reliable software can be developed in this labor-intensive way to efficiently utilize radically new computer architectures such as those now being prototyped.

A related problem is that no "general purpose" parallel architecture can adequately address all problems. Problem-specific architectures can dominate general-purpose sequential or parallel architectures by processing independent problem-components simultaneously and structuring communication paths to minimize data-access or communication delays. However, realization of this potential advantage is notoriously difficult. The cost of developing and testing an architecture is often far greater than the cost of developing and testing software to run on an existing parallel processor.

Problem-specific architecture design can be viewed with software design as part of a more general problem, synthesis of parallel computing systems from problem specifications. Both hardware and software can be derived concurrently to meet performance requirements within technology and cost constraints, or software can be derived to configure and control a specified parallel architecture.

It is argued elsewhere [Green *et al*] that automated transformational development of software from very high-level specifications must play a central role in the paradigm change that is needed to solve the software bottleneck. The argument applies equally well to parallel computing system development.

A knowledge-based transformational refinement paradigm will have several benefits. Algorithm and architecture design expertise, currently acquired and practiced with difficulty by an insufficient number of human experts, will be increasingly codified, leveraged and reused by knowledge-based development tools and systems. Human intuitions will continue to suggest major system design decisions (e.g., specification-refinement rule applications), but automated inference, analysis and simulation tools will provide useful and timely performance predictions to evaluate and guide these intuitions.

8.1.2 Results

The summary and illustration of KTL demonstrate existence and utility of a unified basis for specification of problems, algorithms and architectures. It was not apparent that a relatively simple basis could specify both concurrent algorithms and parallel architectures at an appropriate level of abstraction for use by knowledge-based synthesis tools (which now exist in prototype form or can be developed with known technology).

Computer architectures (including real-time performance parameters) can be fruitfully viewed as parameterized abstract data types with parallel computing systems as instances; these systems may use a rich variety of processing and communication constructs that can be specified naturally in KTL. Preliminary investigations (involving tree, array and connection-machine architectures) indicate that KTL specifications can describe both the architectures and the algorithms to control them. Indeed, they can describe (nonterminating) *reactive* systems that use real-time, concurrency and communication constructs to reason symbolically and interact with their environments.

A precise and understandable semantics is essential for synthesis of reactive parallel computing systems that satisfy their specifications: concurrency, nondeterminism, communicaton and real-time constraints lead to complex specifications that are difficult to analyze and transform to executable form. Knowledge-based synthesizers for such specifications must ultimately be validated semantically on a rule-by-rule basis.

Soundness of inference and transformation rules is well defined in terms of KTL's formal semantics, but effective proof rules are needed to verify applicability conditions of transformations. Proof rules will require further development for practical applications, but the basic theoretical issues and solutions are now well understood. This preliminary work is expected to result in a more detailed formal KTL definition and synthesizer-prototyping plan during 1987.

While much work remains to be done on an appropriate transformational synthesis methodology for KTL, it has been shown that the methodology of [King 85] and related work can be expressed and generalized in this framework. This expression needs to be developed in detail, however.

8.1.3 Organization of this Chapter

Section 8.2 considers what kinds of knowledge must be codified to guide the synthesis of parallel computing systems from very high-level specifications. We distinguish domain knowledge, algorithm-design knowledge, and architecture knowledge, though in fact these realms overlap and require a single wide-spectrum, broad-scope language. We explain what we mean by "spectrum" and "scope" in this context, concluding that no established language or formalism has the required spectrum and scope.

Section 8.3 provides an overview of the KTL specification basis that is being developed at Kestrel Institute. We begin with a general semantic framework for reactive systems, explaining the language primitives on this basis. We explain how such a basis can be "simpler" than a general-purpose manual-programming language such as ADA®.

Section 8.4 illustrates the specification of problems and algorithms in KTL, using parallel prefix-reduction and all-pairs shortest-path problems as examples. We indicate how a transformational synthesis methodology based on KTL could be adapted for either concurrent algorithm/architecture synthesis or algorithm-synthesis for given parallel architectures.

Section 8.5 considers the problem of specifying parallel architectures and computing systems in KTL. An architecture is viewed as an abstract data type, instances of which may contain variables called "stores". Operators of the type constitute the "native language" of the architecture. Real-time constraints on operations express performance properties of architectures. We indicate how tree, array, and connection-machine architectures can be specified in KTL. A*parallel computing system* is an architecture-instance with stores initialized by instructions and data.

Section 8.6 suggests directions for further work, ranging from a system designer's interface with KTL to its interfaces with established specification languages and tools that use them.

Related work is summarized in each section.

8.2. Specification Concepts and Requirements

8.2.1 Spectrum and Scope of Specification Languages

The "spectrum" of a specification language has come to mean the range of abstraction levels that it supports (requirements to code). The "scope" refers to the breadth of domain concepts and constructs that it can naturally express; for example, does it have numeric data types , real-time (clock) and delay primitives, concurrent processing or communication constructs.

The highest abstraction levels (natural-language requirement statements and their more precise formalizations) specify*what* is to be done without specifying how.[1] The *what* may include system or component behavior expressed by mathematical functions or relations; for real-time or "reactive" system specifications it may include performance constraints and communication protocols describing interaction with environmental agents over time.

Lower levels of abstraction specify *how* the specified system or component behavior is to be realized, ultimately at the level of effective operations on data structures. Several well defined abstraction levels and supporting programming styles or formalisms can be identified.

Logic programming lies (in principle) at the interface between "very high" and "high" specification levels: appropriately constrained first-order predicate logic specifications for programs can be "executed" by an inference engine. In practice (e.g., Prolog[]) the severe specification-restrictions (and extra-logical inference-control constructs such as *cut*) necessary to achieve acceptable performance result in *low-level* specifications. Performance requirements thus tend to restrict both spectrum and scope of logic programming.

The promise of logic programming can be (and is being) realized by enhancements of both spectrum and scope coupled with a refinement of the logic-program execution model. We shall see that KTL can be viewed as a realization of this promise; first we consider the needed enhancements and refinement.

Functional programming , like logic programming, specifies algorithms by referentially transparent definitions--in this case, of functions instead of relations. This paradigm is subsumed by logic programming if equality is interpreted as a primitive relation constant and predicates over terms containing function symbols are admitted. By using a many-sorted logic and

[1]We call such specifications *very* high-level to distinguish them from executable specifications written in "high-level" languages such as ADA®.

admitting user-defined data types we obtain a useful executable specification basis typified by **Eqlog[]**.

Concurrent Logic-Programming. **Eqlog** provides no basis for specifying algorithmic control structures or required properties of their real-time behavior. Additional control structure (*guard predicates*) can be imposed on the interpretation of definite Horn clauses to eliminate much of the backtracking (and attendant "low level" control constructs) that have detracted from **Prolog** as a practical specification/ implementation basis. Concurrency is represented by conjunctive goal predicates The resulting formalism (typified by **Concurrent Prolog[]**) has a useful scope as well as spectrum: it can express concurrent systems of actor-like *objects* communicating via streams of messages.Various type systems with inheritance have been proposed to support *object-oriented* programming[] in this context.

Even when the expressive capabilities of **Eqlog** and **Concurrent Prolog** are merged and supported by the powerful logic-program optimization strategies that have been explored[], the resulting formalism is too narrow in both spectrum and scope for the synthesis domain of interest. The spectrum needs extension to "imperative" control structures for efficient execution on existing and future parallel architectures, and the scope needs extension to common real-time and communication constructs.

Temporal-Logics. In recent years it has been realized that "imperative" constructs can be viewed as temporal-logic operators inasmuch as they specify sets of permissable "execution sequences" of system states. Thus sequential composition is an operator on specificatons just as are *next, sometime* and *always*. The **Tempura** language/interpretor **[Moszkowski 86]** and **KTL** are both based on this realization, though their underlying assumptions, primitives, and possibly their ranges of applicability[2] differ significantly. Thus imperative or procedural languages such as **LISP** and **ADA®**) can be subsumed in a temporal-logic specification formalism.

Efficient Execution. Direct interpretation of (extended) logic programs will continue to require powerful inference engines, and processors necessary for adequate real-time performance will in many instances be too costly and complex for critical applications (e.g., in embedded systems). "Smart" compilers or synthesizers that transform logic specifications into efficient

[2]I am indebted to --(oral communication) for useful observations concerning possible translations between Tempura and KTL specifications. Evident differences include Tempura's synchronous "clocked" execution versus KTL's asynchronous concurrency with real-time clock and synchronizing communication primitives. Development of both formalisms in in progress; future in-depth comparisons should prove useful.

programs for existing processors are a step in the right direction. We are investigating their design for KTL and precursers at Kestrel Institute.

8.2.2 Domain Knowledge

There are many important application domains, each characterized by specialized vocabularies, notations, definitions, postulates, algorithms and specialized processing devices. Surely we cannot provide a simple coherent basis on which they can *all* be developed efficiently by declarative and definitional extensions? Perhaps not, but we can hope to identify and support a broad scope of applications with a relatively small basis of orthogonal primitives and combining operators. The choice of basis reflects ones ontology of computations and observable events in digital systems. What, for example, is a process or task (if not a tree of possible future event streams to be interleaved with others), and what does it mean for one task to have higher priority than another? And what exactly is an *event* ?

Important Domain Constructs. Basic domains include numeric and symbolic computation. Real-time applications require durations and times (as provided by real-time clocks) of varying precisions. Digital and software environments provide interfaces that export and import various kinds of entities. Here is a clue: we need't characterize the application domains;[3] we need only characterize the interfaces of objects (including their real-time behaviors) that exist or are to be synthesized for the domains of interest.

Interface Entities. We consider the possible exports of an abstract object. (We will model imports as actual parameters supplied to KTL functions called*object-constructors* .) We view an abstract object x as the tuple of its exports; its type is thus a product-type.

Consider a component x.s. Is x.s a constant or is it a variable? (Alternatively: is x mutable or immutable, and more specifically does x.s vary over time?) If x.s is a "variable", then is it an "out", "in" or "in-out" variable (updated by x or exported operations on x, by users of x.s directly, or both, respectively)?[4]

[3] That is the role of requirements languages and domain knowledge packs to be defined on the specification basis that we provide.

[4]Some (e.g., Parnas) prefer "simpler" interfaces, classifying exports exclusively into V-functions (value-returning functions that have no side-effects) and O-functions (state-changing procedures that return no values). However, we get the effect of an updatable variable x.s by having x.s be a V-function (of zero or more arguments) and exporting a separate O-function x.assign_s. Given that mutual exclusion is supported at a high level by the specification formalism (as in KTL), it seems pointless to complicate the formalism with usage restrictions that can be easily circumvented.

Next we consider the value-type of x.s. (Secondarily, what types are there, and are types "first-class objects" represented by computable values?) There are individual types,(polymorphic) functional types (for V-functions) and procedural types (for O-functions), as well as product and sum types. Types may be construed as subsets of a semantic domain of computable values[MPS]; some of them have internal representations as predicates or retraction maps over the domain, and these may be used as dynamic type-checking or type-coercion functions when appropriate.

Communication Constructs. A particularly relevant kind of interface entity is the *communication variable.* When such a variable is updated, one or more processes awaiting some condition depending on that variable are selected for possible scheduling. It may even be required that some such process be waiting and eligible before the updating process is allowed to continue: this constraint provides a natural model for ADA-rendezvous. This behavior is determined by *synchronization, persistence* and *distribution* properties of the communication variables. Rendezvous is just one of several communication modes to be supported by a specification basis of adequate scope.

8.2.3 Algorithm Design and Architecture Knowledge

Parallel processing has many forms. To design algorithms for a specific architecture it is necessary to specify precisely how computations are to proceed in space (in different processing elements) and time (within what constraints). At a slightly higher level of abstraction it is necessary to specify events to occur in parallel, concurrently (interleaved) or sequentially. Then these events can be allocated to processors, much as virtual registers are allocated to physical registers to optimize resource utilization in compilation of high-level programs.

Automated synthesis of efficient algorithms requires reasonably accurate estimates of performance for different alternative implementations--often before the alternative has been fully refined to a form that is executable on the target computing system. This implies, among other things, that elementary operations must be specified by duration constraints that may depend on data parameters and even (as in switching networks) system load.

Critical applications require specialized parallel processing architectures such as systolic processor arrays or trees that may be sized and shaped to fit specific problem classes. They motivate synthesis wherein both the architecture and the software that controls its computations are derived concurrently from very high-level specifications, VLSI architecture constraints and performance estimations. These have been the focus of work under this contract.

Other applications may use one of several standardized or "general purpose" parallel architectures such as the Ultracomputer[5], Butterfly[6] or Connection Machine.[7] They motivate synthesis guided by efficiency estimates based on a fixed target architecture known to the synthesizer; this will be the focus of a separate research effort.

In both variable and fixed architecture synthesis there is a need to construct and utilize processor-architecture models. Here again the specifications should specify "what" needs to be computed without specifying "how"--perhaps even when sequential implementations are available--because *the best parallel solutions can differ markedly from "parallelizations" of sequential solutions that are familiar to the designer.*

8.3. A Knowledge Transformation Language

8.3.1 Introduction

KTL is an acronym for *Knowledge Transformation Language*. Assertions in KTL transform knowledge (and beliefs) into action; in other words assertions are interpreted as commands or constraints on state-transition sequences. This semantics is the familiar semantics of temporal predicate logic; it unifies the imperative and the declarative programming styles in much the same way as Pratt's dynamic logic and Moszkowski's temporal logic formalisms.

The following is a brief summary; it is not a user's guide or reference manual.[8] We have found it helpful to begin with an overview of the formal semantics of KTL specifications; specific primitives and definable constructs are best explained in that context.

[5]Variants of Schwartz' Ultracomputer are being developed at NYU and IBM (the RP3). This architecture features a few thousand substantial processor-memory elements with global memory segments connected by a non-blocking multi-stage (Omega) switching network.

[6]The BBN Butterfly architecture consists of up to 256 32-bit processors connected by a blocking multi-stage switching network.

[7]The Thinking Machines Connection Machine architecture consists of up to 64K 1×4K bit processor-memory elements connected by a hypercube switching network.

[8]The reader may prefer to read subsequent sections and refer back to this one as necessary.

This version of **KTL** reflects suggestions and critical review by several colleagues at Kestrel Institute; Richard King,Wolf Polak and Richard Jullig suggested significant refinements in the treatment of guarded commands, atomic actions and real-time constructs.

8.3.2 Reactive-Systems : a Semantic Framework

As detailed in **[Pnueli 85]** there are two distinct views (and classes) of computing systems and programs. The first view regards them as functions or input-output relations over states. One may be concerned with the time and storage space required to transform the input to the output, but the details of how this is done are irrelevant to the specification. There is no specified interaction with an environment (other than the input parameters and state), and non-termination (or even its possibility, in nondeterministic systems) is regarded as failure of (total) correctness.

The other view focusses on systems and programs that typically do not terminate and are required instead to maintain a specified (real-time) interaction with an environment. Such systems and programs are said to be *reactive* --an adjective that subsumes *distributed*, *concurrent*, and *real-time* as these features are often essential to satisfaction of reactive-system requirements or specifications.

Reactive-system behaviors cannot be specified by input-output relations; in general they must be characterized by mappings from inputs(initial states) to sets of (possibly infinite) sequences of states and state-transition events, the *possible computations* starting from the input. The sets are one way of representing nondeterministic behavior (or nondeterminate requirements) of reactive systems; branching trees of states (vertices) and transitions (edges) are another (sometimes preferable) way.

We begin with an overview of the objects and relations that constitute an abstract reactive system. We outline an effective *operational* semantics and a compositional *denotational* semantics. The two semantics serve distinct functions but are related, essentially as detailed in **[Plotkin 82]** for a simpler concurrency language. Both must represent and distinguish phenomena known as *divergence* and *deadlock*..

8.3.2.1 Operational Semantics

Configurations of a system are represented syntactically by pairs $<\sigma, c>$ where σ defines a system state and c is a command. A *state* is a mapping from objects called *symbols* to computable values. A *command* is a specification for a set of possible computations, which can be viewed as finite or infinite state-transition sequences.

Atomic transitions of a system are defined by a (generally nondeterministic) relation $\rightarrow$ on configurations and results. *Final* transitions have the form $<\sigma, c> \rightarrow \rho$ where c is an "atomic" command and the *result* ρ is a state or computed value (if c is a value-return command). The transition relation $\rightarrow$ and its reflexive transitive closure $\rightarrow^*$ are defined by structural induction on the structure of KTL commands (below).

The "one-step transition" function determined by a command is given by

$$\text{Trans}[c](\sigma) = \{\rho \mid <\sigma, c> \rightarrow \rho\} \cup \{<\sigma',c'> \mid <\sigma, c> \rightarrow <\sigma',c'>\}.$$

Let S^∞ be the set of finite and infinite sequences of states and results (which appear as final elements only). The *set of possible behaviors* of c on an initial state σ is given by

$$E[c](\sigma) = \{[\sigma1,\ldots,\sigma k,\ldots] \mid <\sigma, c> = <\sigma1,c1> \rightarrow \ldots \rightarrow <\sigma k,ck> \rightarrow \ldots\}$$

where each sequence is either infinite or ends in a result $<\sigma n,cn> \rightarrow \rho$. *E[c]* represents the behavior of c independently of its syntax and is thus a good candidate for the "operational semantics" of c.

Divergence. A command may *diverge* while computing the next state or result. This possibility is represented by transitions $<\sigma, c> \rightarrow \perp$ where $\perp$ represents "undefined". The product-type of configurations is *strict* in the sense that $<\perp,c> = \perp$. Thus $\perp$ is a *result* (albeit undesirable), and we say that $<\sigma,c>$ *diverges* if $<\sigma,c> \rightarrow^* \perp$ or $E[c](\sigma)$ contains an infinite sequence; the latter is of course permissable if c is intended to operate indefinitely.

It may be that c is intended to converge but instead generates an infinite computation sequence (without diverging at any atomic transition). To distinguish this possibility from cases where c always converges we define the "result" semantics of a command c by

$$\text{Res}[c] = \{<\sigma, \rho> \mid <\sigma, c> \rightarrow^* \rho\}$$

$$\cup \{<\sigma, \perp> \mid E[c](\sigma) \text{ contains an infinite sequence}\}$$

where σ ranges over states and ρ ranges over results. Note that Res makes no distinction between divergence (waiting forever) at a single atomic action and divergence (infinitely many convergent actions) of an entire computation; both are equally bad for those who await a defined result.

Deadlock. There is one problem with this semantics that cannot arise in Plotkin's simpler concurrency language: we may arrive at a *deadlocked* configuration $<\sigma',c'>$ from which no further $\rightarrow$-transitions are possible. For example, c' could be a guarded command (below) that awaits an event which never occurs(the guard predicate is false of σ'). We address this problem by defining a "deadlock" state δ and a transition $<\sigma',c'> \rightarrow \delta$ for each deadlocked configuration. Thus δ is a *result* (albeit undesirable), and we say that $<\sigma',c'>$ is *deadlocked* iff $<\sigma',c'> \rightarrow \delta$; similarly, $<\sigma, c>$ *deadlocks* iff $<\sigma, c> \rightarrow^* \delta$.

Even if a reactive system is not required to terminate (or is required *not* to terminate), it is normally required to exclude the possibility of deadlock. Of course a concurrent composition *c1 // c2* (below) may avoid deadlock even though *c1* and *c2* always deadlock: *c1* and *c2* may engage in synchronous communications. Note that nondeterministic commands introduce the possibility that $<\sigma, c>$ may (alternatively) converge, diverge, or deadlock.

8.3.2.2 Denotational Semantics

A denotational semantics for a language is an interpretation $[\![\ ]\!]$ of its operators and expressions into an appropriate mathematical structure or domain. The interpretation must be *compositional* so that, e.g., $[\![c1 \mathbin{//} c2]\!] = \mathbin{//} ([\![c1]\!], [\![c2]\!])$ where *//* is the function denotated by *//*.[9] The critical adequacy conditions of a denotational semantics are known as soundness and full abstraction.

It is clear that a denotational semantics should be *sound* relative to the operational semantics, in the sense that $[\![c1]\!] = [\![c2]\!]$ implies $E[C[c1]] = E[C[c2]]$ for every context $C[\_]$ wherein *c1* or *c2* can occur.[10]

The converse is equally desirable: a denotational semantics should be *fully abstract* in the sense that, if $E[C[c1]] = E[C[c2]]$ for every context $C[\_]$ wherein *c1* or *c2* can occur, then $[\![c1]\!] = [\![c2]\!]$. Note that the operational semantics *E* will *not* be fully abstract for KTL: we may have $E[c1] = E[c2]$ even though *c1* offers a communication (and alternative computation) not offered by *c2*; so long as there is no concurrent process to accept the communicaton, this alternative will not be realized in any state. A context $[c//\_]$ where *c* accepts this communication may nevertheless distinguish *c1* from c2 in the sense that $E[c \mathbin{//} c1] \neq E[c \mathbin{//} c2]$.

A denotational semantics for commands should satisfy additional requirements. The requirement of compositionality suggests that $[\![\ ]\!]$ should model the one-step transition function *Trans*. This can be accomplished by interpreting commands into a domain *R* of *resumptions* that satisfy an isomorphism

[9]The operational semantics need not be compositional in this sense: E provides a "meaning" for commands such as [c1 || c2] without assigning a meaning to ||.

[10]A *context* C[_] is an expression schema with a single "slot" _; C[e] is the expression obtained by replacing _ with e.

$$R \cong [\mathit{State} \rightarrow \mathbf{P}[\Delta^{\eta} + [\ \mathit{State} \times R]]]$$

where $State = [Symbol \rightarrow \Delta^{\eta}]$ (a domain of maps from *Symbol* to a domain Δ^{η} of values) and $\mathbf{P}[\Delta^{\eta} + [\ State \times R]]$ is a domain of possible one-step transition results (returned values and *<state, resumption>* pairs). The intuition is that this isomorphism interprets a resumption as a map from states to sets of possible outcomes. We have extended the work of **[Plotkin 82]** to solve this equation so that *R* and *State* are both retracts (subdomains) of a "universal value domain" Δ^{η}, an extensional model of the untyped λ-calculus with constants representing KTL primitives.[11] Consequently it is easy to interpret *procedures* $\lambda x.c$ (where *c* is a command) as functions in the retract $[\Delta^{\eta} \rightarrow R]$ of Δ^{η}.

The domain $\Delta^{\eta} = <\Delta^{\eta}, \sqsubseteq, \subseteq>$ has an approximation ordering $\sqsubseteq$ and a semilattice ordering $\subseteq$ that represents "degrees of nondeterminism". Intuitively, $[\![e]\!]$ represents the set of all possible results of evaluating *e*, which may be a nondeterministic construct in a language such as KTL.

Correctness-Preserving Refinements. The $\subseteq$-relation is essential for defining *correctness* of specification refinements: a refinement of s to s' is said to be *correct* provided that $[\![s']\!] \subseteq [\![s]\!]$; that is, the possible behaviors of *s'* are included among those of *s*. An implementation is not required to preserve *all* of the nondeterminism admitted by a specification (this would be preservation of *meaning* instead of *correctness*), but its alternative behaviors must all be admitted by the specification.

To summarize, an adequate denotational semantics for specifications characterizes the relation of *behavioral equivalence in all contexts*, also known as *observational congruence* [], by means of a compositional meaning function, and it characterizes *correctness of specification refinements* in terms of a semilattice-inclusion relation on set-like values that represent nondeterministic functions and results of nondeterministic operations.

8.3.3 Terms, Commands and Procedures

8.3.3.1 Commands as Relations

KTL views each command has an event predicate; assertion of this predicate is synonymous with execution of the command. Thus the imperative "do*this*" is read as an event description, "*this* happens"; $x:=e$ is a relation that holds in a state-transition event "x gets value of e". *q.enq(e)* is a

[11]Work in progress.

relation that holds between *q* and *e* when the *enq* command is interpreted or executed: in the next state, *e* is at the back of the queue *q*. In this way an imperative language can be viewed as a temporal predicate-logic specification language. KTL also has higher-level specification constructs [Section 8.3.6]; placing all in the same logical space of event descriptions is a key step in their definition and use.[12]

8.3.3.2 Syntax

There is a syntax of terms and a syntax of commands. For terms it is necessary to know, e.g., that v.p is the p-component of a tuple v having named product type, and that sentences are closed Boolean-valued terms. Here we just summarize the syntax of primitive commands.[13]

```
acmd ::= [ skip ][14] | assignment | assertion | return
assignment ::= var := term | « vars » := term
assertion::= sentence[15]
gcmd ::= acmd | icmd | dcmd
icmd::= sentence ? cmd [;; cmd] | sentence ? dcmd[16]
dcmd ::= sentence ! cmd [;; cmd]
ascmd ::= gcmd | gcmd | ascmd | [ ascmd]
qcmd ::= ascmd | ∃ vars . qcmd | ∀ vars . qcmd
scmd ::= qcmd | qcmd ; scmd | [ scmd]
ccmd ::= scmd | scmd || ccmd | [ ccmd]
cmd ::= ccmd | µvar . cmd | procedure ( arguments)
```

[12]This view of imperative constructs provides a more familiar basis of temporal logic specification constructs than that developed in **[Mostowski 85]**; it is also significantly different, as Mostowski's approach is incapable of expressing or admitting the nondeterminism that follows necessarily from asynchronous concurrency. This view will be developed in a separate note.

[13]We reserve brackets for grouping commands and terms, using «...» for tuples and sequences, {...} for sets and mappings. **Boldface** indicates terminal symbols.

[14]**skip** may be omitted: (non-bold-face) brackets indicate optional constructs.

[15]**return** is almost needed as an explicit operator to return a value from a command to distinguish between asserting x=0 and returning the current truth value of [x = 0]. But in fact there is no actual ambiguity since we distinguish between commands and value-returning forms that must end with a value of specified type to be returned. *[[x=0 ? true | x≠0?false]* makes it apparent that a Boolean is being returned. An assertion *[x=0]* awaits a concurrent action leading to a state wherein x=0 and then continues.

[16]Note the distinction between *p?q! a;;b* and *p?[q! a;;b]* : the latter is an icmd (interrogative-[guarded] command) with action *[q! a;;b]* that must follow verification of the guard *p* indivisibly; *p?q! a;;b* is an "interrogative-declarative" guarded command with action *a* and sequel *b* ; its guard must verify *p* and then (indivisibly) assert *q*.

Other commands are defined on this basis; e.g., ***let decls where constraints in scope*** declares some local symbols and maintains some contraints among them in *scope*; it is defined by a qcmd

$$\exists \text{ dvars. [defns} \wedge \textbf{always} \text{ (constraints) ! ;; scope]}$$

where *dvars* are the locally defined symbols and *defns* is the initialization of the independent ones. Likewise the rule body $\forall \underline{x}.pre \dashrightarrow \exists \underline{y}.post$ abbreviates

$$\forall\, \underline{x}.\text{[pre ?;; sometime } \exists \underline{y}.\text{post] .}$$

The usual iteration constructs (***for iterator do cmd, while sentence do cmd***) are defined similarly. A *procedure* is just a λ-abstraction $\lambda \underline{x}.c$ where c is a command.

8.3.3.3 Informal Semantics

atomic commands. **skip** is the identity command; it is a conventional synonym for **true** in **KTL** (as asserting **true** is a no-op). *Assignment* updates the specified variable(s) with the current value of the expression. An *assertion* updates the current situation to satisfy the asserted sentence; synthesis may be required to make the assertion executable. A *return* terminates the responsible task and returns the specified value.

guards : **otherwise, delay**(*duration* [**after** *time*]), boolean expressions that may involve **priority, now** (the current time), and communication relations (P $\underline{m}$) where P is a communication-predicate variable.

dcmd. A declarative-guarded command <u>*assertion* ! *action*</u> ;; *cmd* waits until the *assertion* can be consistently asserted ; then *action* (a command) is performed indivisibly in this extended state, and subsequently *cmd* is performed (allowing intervening actions).

icmd. An interrogative-guarded command <u>*query* ? *action*</u> ;; *cmd* waits until the *query* is true in the current state and then indivisibly performs the *action*, continuing subsequently with *cmd*.

The *query-assert* form <u>*query* ?*assertion* ! *action*</u> ;; *cmd* awaits a state in which *both* the *query* is true and the *assertion* can be made without contradicting current invariants, then indivisibly (asserts the *assertion* and then performs the *action*), continuing subsequently with *cmd*.

The existentially-quantified command ∃ *bvars* . *ascmd* awaits a state in which there can be found bindings of the *bvars* (of specified types) that enable one of its guards to be satisfied, then indivisibly performs the atomic prefix of the selected *ascmd* alternative, then completes the alternative(still with these bindings).

There are restrictions on the kinds of variables and guard contexts in which existential quantification is effective. In particular, if the guard involves communication relations then the sender (declarative-guarded command) and receiver (interrogative-guarded command) must have communication predicates whose unification is a ground (closed) atomic formula. This unification determines the bindings of existentially quantified variables for both sender and receiver(s).

The "enumerate for all" construct $\forall w.G?A;;C$ concurrently finds all w such that G holds and does A;;C for each; its extension to alternative-selections is straightforward. There should be only a finite set of values for which the alternative selected results in a non-skip action, and the different values should result in non-interfering computations.

The alternative-selection $[gcmd_1 \mid ... \mid gcmd_n]$ awaits a state wherein one of its alternative guards can be satisfied; one such alternative is then selected and executed. An **otherwise** (interrogative) guard is true iff none of the other alternative guards is true; a (**delay** *duration* [**after** *time*]) predicate is true forever after *time* + *duration* ; e.g., (**delay** *duration*) is true *duration* after *now*, where *now* (the default *time*) is the time when the predicate is first evaluated in the alternative-selection process.

If one of the guards is true, or can be made true by a consistent declarative-guard assertion, then the alternative-selection may proceed without suspending the task that executes it.

The concurrent composition $[\ cmd_1 \parallel ... \parallel cmd_n\ ]$ suspends the task that executes it and creates n new concurrent tasks. The suspended task resumes when (if ever) each of the n new tasks has completed. A declarative-guard prefix of each cmd_i may specify a constraint on **priority** for the extent of the new task specified by cmd_i ; the new task will be executed with a priority that satisfies the asserted constraint.

Recursion. The form $\mu id.form$ can be used to form recursive commands, subprograms, or data types.

8.3.4 Communication Variables

Communications as Relations. **KTL** embodies a theory of communications as events that update knowledge states. Two tasks may communicate by updating and querying a shared resource such as a queue or buffer (with appropriate mutual exclusion), or by updating and querying a shared knowledge state, specifically, by asserting, querying and retracting relations. The *shared predicate variables* involved in these interactions have attributes that determine communication protocols among the processes that use them (below).

Predicate variables serve as "communication ports" between the processes that update and query them: one process asserts (P m) (*sending*), another finds m such that (P m) is asserted (*receiving*); in doing so the receiver may *retract* (P m). Thus the processes actually communicate via P, a special kind of shared variable that can only be updated by assertion and retraction of message communications.

Applications require a variety of communication protocols, and so a variety is found in various real-time and concurrency languages. An examination of these motivates a theory based on three almost orthogonal attributes of communications or, more precisely, communication-predicate variables. These attributes are *monitoring, synchronization, persistence*, and *distribution..*

Summary of Attributes

Monitoring. A Boolean or predicate variable may be *monitored* or *unmonitored* :

> Monitored--concurrent task scheduling operations when a monitored Boolean or predicate-variable relation is asserted, according to the communication attributes described below.
>
> Unmonitored-- an ordinary logical relation, whose assertion does not affect scheduling except insofar as it may make true certain guards of suspended tasks (suspended at a guarded command or alternative-selection) . There is no guarantee that such a task will be activated just because its guard happens to become true for a while; the scheduler need not notice every such event.

The following attributes govern the treatment of monitored variables:

Synchronization. A communication may be *synchronous* or *asynchronous*.

> Synchronous-- sender blocks(waits) until receiver(s) receive it; there must be at least one receiver. Sender(asserter) and

receiver(s)(queryers) combine to form a single (atomic) action; (P m) is retracted as a part of this action.

Asynchronous--sender does not block: assertion does not require any receiver(s).

Persistence. A communication may be *persistent* or *transient* :

Persistent -- relation holds after it is asserted, unless explicitly retracted by a task.

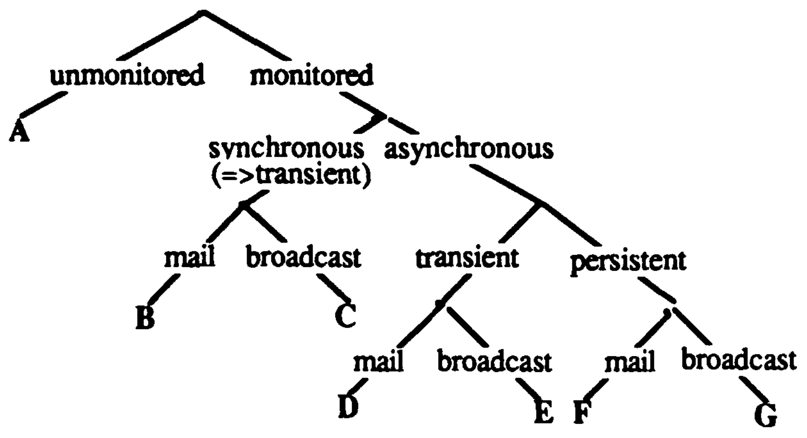
Transient -- relation is retracted automaticlly in the next state following the communication event. In effect, the relation is true of the *atomic transition* between two states.

Distribution. A communication may be *mailed* or *broadcast* .

Mail--point to point: one receiver, which then retracts the relation(to prevent others from receiving the same message).

Broadcast--all receivers that await a compatible message receive it.

Not all combinations of these attributes make sense, inasmuch as *synchronous* implies *transient* (to prevent querying the same relation by subsequently waiting tasks, which would violate synchrony):



Attributes Of Predicate Variables

The reader will be familiar with at least some of the possible combinatoins of these attributes:

A. ordinary Boolean and predicate variables.

B.*rendezvous*, as used in Ada or CSP (but ours admits the symmetric decoupling of Gelernter's).

C. an all-waiting-receivers rendezvous, at least one receiver required.

D. a communication that is detected (by one task) iff some task is awaiting it, and is otherwise lost.

E. *pulsed-event* in French real-time language LTR3, though restricted there to Boolean variables (pure signal, no message). An all-waiting-receivers rendezvous, no receivers required. Differs from B only in that it cannot block the sender.

F. mail that persists until some task (and only one) receives it.

G. *event* in LTR3; broadcast to every task that awaits it. Remains true until explicitly retracted (by asserting *next*($\neg e$)).

Remarks

1. Retraction of the communication relation, where specified above, is an update of the Boolean or predicate variable that *automatically* accompanies detection and reception of the message by the receiver(s). *This retraction is a part of the semantics of communication variables that have attributes* (mail, transience) *which imply retraction.; i.e., it is implied by the language-defined communication operations.*

2. Declaration of an attribute for a communication variable amounts to declararing a restricted subtype[cf. Appendix]; we may introduce corresponding modes, e.g., SM (τ -> boolean) (Synchronous Mail) for rendezvous-predicate variables of value-type (τ -> boolean).

3. Rendezvous in CSP and Ada differ in that different classes of communication relations are asserted and queried: in CSP the receiver must name the sender; this amounts in KTL to requiring that a unique sender-identifier be an explicitly specified argument of the communication relation queried by the receiver. In Ada only one intended receiver can be named. In KTL neither, one, or both can be required; the programmer chooses. The "sender" asserts a relation (P $\underline{m}$) and the "receiver" queries a relation $\exists$ $\underline{x}$. (P $\underline{e}$)? ... where $\underline{e}$ is an appropriate expression built up from bound variables of $\underline{x}$ and other defined symbols and literals. The communication occurs only if (P $\underline{e}$) matches (P $\underline{m}$), and then ... is interpreted or executed with the bindings for $\underline{x}$ determined by this matching.

4. Synchronous communications among tasks with priority follow Ada's "maximal priority" rule for the actions (read: entry bodies) that indivisibly

follow the communication guards: the highest priority of a participating task determines the priority of the rendezvous. |

Related Work. Perhaps closest to the present proposal in viewpoint and functionality is the communication facility embodied in David Gelernter's *Linda* language and system.[17] Gelernter's language supports communication based, in effect, on atomic assertions, queries and retractions of communication relations (labeled tuples) in a global distributed message space. Matches between senders and receivers are based on unification of corresponding fields in message patterns: a match occurs when sender and receiver together have consistently specified all fields. This corresponds closely to the asynchronous, persistent broadcast communication mode described below. Gelernter shows that other communication protocols can be developed on this basis. But Gelernter's is only one of several primitive communication modes supported in KTL.

3.5. Object Types and Instances

Kinds of Objects. We consider three increasingly rich kinds of types definable by the same basic mechanism in KTL:

- *algebraic data types* whose instances are best construed as values assigned to variables.
- *mutable-object* types whose instances are variables updatable by procedures defined by the type.
- *active-object* types whose instances contain processes that execute concurrently on their behalf.

An abstract object x is viewed in KTL as a tuple of values (typically functions and procedures) that may depend on the internal state of x. Thus x has product type $\mathbf{prod}(o_1{:}\tau_1,\ldots,o_n{:}\tau_n)$ and $x.o_i : \tau_i$ is its *ith* component. Component projections $o_1,\ldots,o_n$ are the exported or visible operators of x's type.

Active-object types constitute a powerful modelling tool. They can be modelled in Ada by tasks and generic packages whose instances contain tasks. They should not be introduced lightly: an active object can mutate spontaneously and can communicate with tasks that await or assert events involving communicaton variable that the object exports.

Given an appropriate type and type-inclusion theory, and primitives for extending types and their instances with new visible operators, we can

[17]Gelernter, David, *Generative Communication in Linda,* ACM TOPLAS 7(1) (Jan 85), 80-112.

develop in this way a disciplined object-oriented programming style comparable in expressiveness to **Smalltalk** and the **ZL Flavors** system, but more safely and at a higher level of abstraction.

Example

Elsewhere we make use of the type constructor *port*. An object p = port(τ) can be initialized by p.store(e) where e: τ ; only then can we apply p.fetch(y) where y: var(τ). This "empties" the port until another store-operation is performed. A "reset" operator empties the port without assigning the stored value.

```
port = λ τ :type.
        let v: var(τ ) = new(undefined),       -- v to be initialized by first
store-op
           store:SM(τ ->Bool),                 --Synchronous Mail, or
           fetch:SM(var(τ )->Bool)     --rendezvous
           reset: SM(Bool)             --variables;
        in «store::,fetch::, reset::, μ L.[[∃ y:τ .store(y)?v:=y];
                                          [ ∃ z:var(τ ).fetch(z)?z:= ↓ v ;; L
                                          |          reset ? ;;L]»
```

Thus each instance of port(τ) behaves like a singly-buffered port of type τ . The expression states'port(τ)' denotes the set of all states of instances of port(τ); a declaration

```
        let p: states'port(τ )' = port(τ ) in  scope
```

binds p to a new instance p of port(τ) in scope, where commands p.store(e) and p.fetch(y) may occur; the *: states'port(τ)'* part is optional by type inference. Note that the final (process) component of a port is "hidden": its projection operator (selector) is not specified.

8.3.6 Higher-Level Specifications and Constraints

A quantified temporal logic has been developed for specification of KTL program units. During synthesis of a program, the guards of commands and commands themselves can be temporal logic predicates, which constrain the possible computation trees generated by the action and resumption. The *next-time* operator designates the state following the action. Guards of commands within complex actions may also contain next-time operators; these are interpreted at the next-lower architectural level, which is not visible outside the action. This logic is well suited for hierarchical software development; it includes the unified linear-branching time logic CTL* of Clark and Emerson and can specify constraints on a computation's history as well as its possible futures. We are investigating possible uses of this logic and its sublogics in synthesis and verification of concurrent real-time programs and computing systems. A summary is given below. The logic is based on the operational semantics for KTL programs.

A sequential interleaving of concurrent tasks is said to be (*weakly*) *fair* if the interleaved computation is infinite and every task that is almost always ready is allowed to make progress infinitely often. In the presence of guarded commands and priorities, we say that a task is *ready* provided that its next statement's guard, if any, is true, and its priority is highest among all such tasks. A task is *almost always* ready if, after some finite prefix of the computation, it is ready forever afterwards.

Note that we do not count access to shared resources among possible reasons for a task not being ready; as noted above our language is based on primitive and defined atomic actions, not on access to shared resources.[18] The definition of "ready" is based on snapshots of the computation's configurations *between* actions, when all lower-level access to visible resources has been released.

Temporal modalities are defined in terms of *situations*(occurrences within commands of configuragions) that are accessible from the current situation by a language-defined transition relation Alt. This relation defines a tree: if Alt[ρ, σ] and Alt[ρ', σ] then $\rho = \rho'$; Thus the *finite linear history* of each situation is accessible via a map Prev: Sit–>Sit.

Definition. A *frame* is a system [Sit, Alt, ...] where Alt is a binary relation over Sit; Alt defines the transition relation of KTL commands[8.3.2].

We are concerned with *specification* guards G?A;;C or G!A;;C where G is a temporal logic sentence using the operators defined below. G is interpreted in the frame of a computation tree at a situation (the origin) where the specification-guarded command is encountered. G is interpreted over the computation subtree of situations reachable by executing A;;C (and any interleaved concurrent processes). If G is valid over this computation

[18]In any event the patterns of access necessary to implement our actions cannot possibly introduce deadlocks. Of course locks can be implemented and tested by guarded actions, and their misuse can introduce deadlocks, but these are explicit software-defined guards that do not affect the above definition of weak fairness.

subtree then G?A;;C = **true?**A;;C. Validity is sometimes expressed by , e.g., [**true?**A;;C] **sat** G.

Declarative temporal-logic constraints G!A;;C will be used used by the synthesizer to transform [true ? A;;C] into an implementation [true? A';;C'] such that [true? A';;C'] **sat** [G ? A;;C].

Modalities for Computation-Tree logic with History (CTH):

State Modalities: the following operators construct *state formulas*, which are true or false of a situation, called the ***origin*** of the formula.

∀_path	∀_path'B' holds iff B holds over every possible path (starting from the origin of the assertion)
∃_path	∃_path'B' holds iff B holds over some possible path (starting from the origin of the assertion)

Path Modalities: the following operators construct *path formulas*, which are true or false of complete computation paths rooted at the original situation where the formula is asserted or queried. Each path is a non-null finite or infinite sequence of situations. Each situation in a path can be traced back through its predecessors and possibly beyond to a unique *initial* situation by means of the Alt-converse relation.

init[ially]	init'B' ⇔ B holds in the initial situation
orig[inally]	orig'B' ⇔ B holds at the origin of the assertion or query
next	next'B' ⇔ B holds in the next situation (implies existence of a next situation)
prev[iously]	prev'B' ⇔ B holds in the previous situation (implies existence of a previous situation)
fin[ally]	fin'B' ⇔ B holds in the final situation, for a finite path; or B follows from the assertions made and never retracted, for an infinite path.
unless	['B' unless 'C'] ⇔ [C ∨ [B ∧ (next'true' ⇒ next'['B' unless 'C']')]]
until	['B' until 'C'] ⇔ ['B' unless 'C'] ∧ ∃_fut'C'
since	['B' since 'A'] ⇔ [B ∧ A]∨ [¬B ∧ ∃_past'A' ∧ prev'['B' since 'A']' --B has held ever since A was last true.
;	['A'; 'B'] holds in a path $<\sigma_1\ldots>$ iff there exists i such that A holds in $<\sigma_1\ldots\sigma_i>$ and B holds in $<\sigma_i\ldots>$. *;* is also known as *chop*. [Pnuelli, Logic in CS 86]
∀_fut[ure]	∀_fut'B' ⇔ B ∧ [next'true' ⇒ next'∀_fut'B' ']
∃_fut[ure]	∃_fut'B' ⇔ B ∨ next'∃_fut'B' '
∀_past	∀_past'B' ⇔ B ∧ [prev'true' ⇒ prev'∀_past'B' ']
∃_past	∃_past'B' ⇔ B ∨ prev'∃_past'B' '
∀_time	∀_time'B' ⇔ ∀_past'B' ∧ ∀_fut'B'
∃_time	∃_past'B' ∨ ∃_fut'B'.

These provide a more expressive temporal logic basis than the simple linear-time operators; they extend the computation-tree logic of[EmHa] to support specifications that refer to computational history of situations. Commonly used modalities are defined on this basis below.

Definition(State and Path Formulas summarized)

State formula:

Every atomic formula is a state formula.

If p and q are state formulas then so are $p \wedge q$ and $\neg p$ (etc.)

If p is a path formula then ∀_path'p' and ∃_path'p' are state formulas.

Path formula:

Every state formula is a path formula.

init'p', orig'p', prev'p',next'p', fin'p', 'p' until 'q', 'p' unless 'q', ['p' since 'q'], ['p' then 'q'], ∀_fut'p', ∃_fut'p', ∀_past'p', ∃_past'p' ∀_time'p', ∃_time'q' are path formulas.

Convention. A path formula p is interpreted in a situation σ by prefixing it with ∀_path.

Real-time temporal operators are easily defined from the above, the real-time clock function **now**, and the real-time predicate **delay** where

(**delay** dur **after** tim) is true forever after tim + dur,
(**delay** dur) is true forever after **now** + dur.

Most common are operators that specify some condition will hold from **now** to **now** + *dur*, or some condition will hold within this interval:

∀_fut_within(d,B) ⇔ ∀_fut ' B ∨ (**delay** d **after orig'now'**) '
∃_fut_within(d,B) ⇔ ∃_fut ' B ∧ ¬(**delay** d **after orig'now'**) '

These are path formulas. The formula

∀_path ' ∃_fut_within(d,B) '

is valid in a situation ρ iff, for every path of situations descending from this in the computation tree rooted at ρ and representing the complete execution of ρ's command, there exists a situation ρ' wherein

$(\rho' \models B)$ and $(\mathrm{now}(\rho') - \mathrm{now}(\rho) \leq d$.[19]

Sometime and always. More convenient state formulas are defined by

[d]B ⇔ ∀_path '∀_fut_within(d,B) '
<d>B ⇔ ∀_path '∃_fut_within(d,B) '.

These are the duration-bounded versions of sometime and always:

always'B' ⇔ ∀_path '∀_fut'B' '
sometime'B' ⇔ ∀_path '∃_fut'B' '

8.4. Problem and Algorith Specification

8.4.1 Problem Specifications

The following problem statements illustrate what we mean by *very* high-level specifications (the input-output behavior level) as opposed to *merely* high-level (the algorithm level of today's high-level languages). We develop their solutions subsequently; here we argue that these solutions can be realized in a variety of increasingly general architectures ranging from problem-specific VLSI systems to programmable parallel architectures. All are instances of the parallel computing-system synthesis technology described below: constraints on the target architecture can be embedded in the problem statement.

All-prefix summation . Given an input vector X of elements with a commutative associative operator +, the problem is to compute an output vector Y such that size(Y) = size(X) and $Y(k) = \Sigma\, [X(i) : 1 \leq i \leq k]$. It is not difficult to synthesize an ordinary linear-time algorithm for this commonly occurring subproblem using available automated synthesis methods; the specification $[\Sigma\, [X(i) : 1 \leq i \leq k] : 1 \leq k \leq \mathrm{size}(X)]$ (with only minor notational change for summation) compiles quickly to reasonable CommonLISP code in the Kappa synthesis system being developed at Kestrel.

But linear is often too slow. In a typical application the input vectors arrive in parallel and the prefix-sum vectors are accessed in parallel; anything more than ***constant*** time to compute the prefix-sum vector is therefore a computation bottleneck. Since the last prefix-sum element is the input-vector sum, it is clear that an O(lg size(X)) time solution is best-possible. Here the strategy of adapting a recognized subproblem-solution to solve the main problem wins. A divide-and-conquer design strategy leads to log-time

[19]$(\rho' \models B)$ iff B holds in the rooted Kripke structure defined by ρ'; if B contains no modalities then this is equivalent to B holding in the state defined by ρ'.

solutions with binary processor-tree architecture. The internal- and leaf-node algorithms can be realized as software or compiled to VLSI. |

All-pairs shortest-path. A directed graph or *digraph* is a pair (V,E) where e: V ->V is a partial function ($e \in E$). A *path* from v to v' is a sequence $[e_0,\dots e_{n-1}]$ such that $e_{n-1}\bullet \dots \bullet e_0(v) = v'$. A *cost function* for (V, E) is a map $c: E \times V \to R^{\infty}$ such that $e(v) \in V$ implies $0 < c(e,v) < \infty$ and $e(v)\uparrow$ (undefined) implies $c(e,v)= \infty$. We extend c to $\underline{c} : E^* \times V$ by

$$\underline{c}([e_0,\dots e_{n-1}],v) = \Sigma_{k=0}{}^{n-1}\ c(e_k,e_{k-1} \bullet \dots \bullet e_0(v)).$$

The *distance* d(v,v') is then given by

$$d(v,v') = \min\{\underline{c}(\underline{e},v): \underline{e} \in \text{Path}(v,v')\}$$

Given a finite digraph (V, E) and cost function c the problem is to compute d so that d(v,v') can be determined in constant time ($v,v' \in V$).

Dynamic programming and store-versus-recompute strategies lead to a linear-time solution with a mesh-connected processor array containing $|V|^2$ elements. Again there is a choice of realizations for the constituent array-element algorithms. |

In both of these examples, sophisticated analysis is needed to verify that data arrives when and where required. In other words, *automatic verification that simple problem transformations preserve correctness or meet performance constraints is at or beyond the state of the art* . Interactive analysis and inference support may be more appropriate than full automation.

8.4.2 Algorithm Derivations

While parallel solutions for the prefix and shortest-path problems are known, their automated synthesis from behavior specifications and constraints is a very difficult problem that we are just beginning to understand. Richard King has developed a theory of synthesis based on units of computation called closures, and has applied it manually to the derivation of solutions for such problems[King 85].

These derivations are guided by declarations and transformations of processor-data structures that eventually model the synthesized architecture. However, rather than rely on a few *ad hoc* architectural structures, it would be useful to see them as instances of a general architectural modelling capability. An architecture should be viewed as an abstract-object type, and a parallel-computing system as a problem-specific instance of this architecture. Instance parameters may be simple problem-size specifications (for fixed-program architectures), or they may include element-algorithms (for programmable architectures).

Approach.From the all-prefix summation problem we derive an architecture *Tree* such that a declaration *P: Tree(lo,hi)* yields a binary tree-structured concurrent program with leaf-vertices numbered lo,lo+1,...,hi, each with an input and an output port. The abstract-object type*Tree* may be parameterized by element type and operator as well as low and high leaf-indices. *P* may be compiled to CommonLISP for rapid prototyping in the Kappa environment, or (given suitable extensions) compiled to a VLSI specification in a silicon-compiler input language.

From the all-pairs shortest-path problem we derive an architecture Mesh such that Mesh(n) is an $n \times n$ mesh-connected architecture. A declaration P: Mesh(n) yields a concurrent process array whose (i,j) element can be initialized to the minimal cost c_{ij} of an edge from v_i to v_j by a message $c(i,j,c_{ij})$. |

These are realistic self-clocked solutions. They can be saved in a library of architectural types for instantiation where needed in VLSI systems. Instead of generating a fixed problem-specific VLSI cell for each processor element the architectures may be further parameterized with a program for each processor, leaving only the architecture's shape and connectivity invariant.

This *programmable-architecture* approach yields classes of machines with regular interconnection and communication paths between elements. Each processor-memory element has reasonably general program-execution capabilities tailored to its interconnections with neighboring elements. Each machine can then handle a family of parallel algorithms for problems that fit its capacity and architecture. The prefix-summation algorithm is obtained from a programmable Tree architecture by supplying a generic program for each interior vertex, and a procedure Leaf such that Leaf(k) controls the k-indexed leaf. The all-pairs shortest path algorithm is obtained by supplying to the abstract Mesh(n) architecture a procedure M such that M(i,j) has the appropriate i,j-element behavior. Architectures such as DADO and programmable systolic arrays are typical of this approach.

We want a device to compute the function

$$\lambda\ X{:}seq(real).[\Sigma\ [X(i) : 1 \leq i \leq k]: 1 \leq k \leq size(X)].$$

The synthesis system prototype can already compile essentially this specification to LISP within a couple of seconds, though without further guidance and optimization the solution found is an $O(size(X)^2)$ sequential one.

So we need to be more specific. We want Kappa to synthesize a parallel computing architecture*Tree* such that a declaration *P: Tree(lo,hi)* yields a binary tree with nodes leaf(P,k) for k $\in$ lo...hi, each with an input and an output port. *P* responds to a *start* signal by prompting its leaves to accept the next input and invalidating any previous data that may be in its output ports. Once filled the output ports are valid until the next external *start* signal; we want to specify that the outputs are valid within

$$O(\lg (hi - lo))$$

time units of this signal. The name *Tree* is leading but irrelevant here: we are only specifying its interface; Kappa is supposed to find the architecture.

Given P as above, we suppose P has ports Leaf(P,k).in and Leaf(P,k).out of type **real** for all k $\in$ lo..hi; for each such port p, $\downarrow$ p is either a real number or is **undefined** at any point in time; an attempt to fetch $\downarrow$ p when p is **undefined** causes the request to wait until p is defined.

Now we can refine the specification by

$\forall$ k: lo..hi. $\exists$ v:real.[$\downarrow$ Leaf(P,k).in = v] $\wedge$ Start(P) =>
    sometime-within(c • lg(hi -lo),
        $\forall$ k: lo..hi. [$\downarrow$ Leaf(P,k).out =Σ[$\downarrow$ Leaf(P,j).in :
(j)lo$\leq$j$\leq$k])

where c is a positive real constant independent of lo, hi. The call-predicate Start(P) = [P.up.reset; P.start; P.start; P.start] for P is explained below; it could be simplified by using a root node type different from other internal node types.

This is the constraint that Tree must satisfy for all lo<hi. In [King 85] a method for applying divide-and-conquer transformations to specifications such as T's is developed; it provides a systematic basis for solving recursive definition schemas such as the one given for Tree below.

8.5. Architecture and System Specification

Section 5.1 describes the result of transforming the parallel prefix algorithm into a parallel computing system. Section 5.2 summarizes some correspondences between KTL constructs and hardware constructs. Tables of such correspondences can be used to terminate refinements in synthesis of a hardware device from specifications.

8.5.1 Parallel Prefix System

8.5.1.1 Tree and Interior Node Types

Let mid(lo,hi) = lo + (hi - lo)/2 (integer-division).

```
Tree(lo, hi) = [lo = hi ? Leaf(lo)
             | lo < hi ? DC(Tree(lo,mid(lo,hi)), Tree(mid(lo,hi)+1,hi)) ]
```

DC is a parameterized interior-node type constructor:

```
DC = λ L: instance(Tree(*,*)), R: instance(Tree(*,*)).
     Let  u: var(real) = new(0),
          v:  var(real) = new(0),
          up: port(real) = new,
          start: SM(Bool)             --synchronous mail signal
     in
     «up::, start::, Left:: L, Right::R,
     μ S.[start?;;L.start;R.start;     --receive, propagate external start
          L.up.fetch(u); R.up.fetch(v); --subtree sums
          up.store(↓ u + ↓ v);     --propagate sum to parent
                start ? ;;          L.start; R.start;
                --propagate increment-cycle start

          R.up.store(↓ u);
          --increment R subtree by  ↓ L.up
                μ C.[ up.fetch(v)!;; L.up.store(↓ v); R.up.store(↓ v);
                  C
                       --propagate increments from parent
                    | start?;;L.start; R.start; S]  --until done; set out-
                                                   --ports
        ]                                       --and return to initial state
     »       --exports
```

Note that *start* is used three times: once to initiate the computation from filled in-ports of leaves, once to initiate a cycle of receiving and propagating increments from the parent node, and once to terminate this cycle and ultimately set the out-ports.

Consequently the proper calling sequence for P is

reset(P.up); P.start; P.start; P.start.

This sequence can be simplified by using a root node that is slightly different from the other interior nodes; for the root node P.up is superfluous and there is no parent to propagate increments downward.

A Boolean port could be used instead of the mail signal *start* ; the mail signal is somewhat simpler to use, in part because it is automatically retracted on receipt. A transformation rule can be added to automatically replace a synchronous mail signal used only for communication between two tasks with a Boolean port.

8.5.1.2 Leaf Nodes

Given P: Tree(lo, hi) we can define leaf(P): lo..hi -> instance(Leaf(*)) by

```
                leaf(P, k) = if lo = hi then P
                             else if k ∈ lo..mid(lo,hi) then leaf(P.left,k)
                             else leaf(P.right, k).
```

This is the leaf-selection function for a given tree; in a real computing system it may be realized so that information can be fed to leaves in parallel.

The parameterized type constructor Leaf is defined by

```
Leaf = λ k: Natural.
        Let   v: var(real)=new(0),
              up: port(real) = new,
              start: SM(Bool),
              in: port(real) = new,
              out: port(real) = new
        in
         [μ S.[ start ?;; reset(out); in.fetch(v); up.store(↓ v); --
fetch,propagate in-port
              start ?;; μ C.[ up.add(v)!;; C              --add increments
                            | start?;;out.store(↓ v); S]  --until completed;
output.
              ]
        ||«up::,start::,in::,out::»]
```

Note that *start* is used three times: once to reset the out port and read the next input, once to initiate a cycle of zero or more increments from the parent, and once to set the out port and await another external *start*.. The parent node ensures that these signals arrive at the appropriate times.

8.5.1.3 Correctness

The argument that P: Tree(lo, hi) satisfies the specification of Section 5.1 proceeds by induction on size(P) = hi - lo. It relies on the fact that the sum of each subtree is propogated to its parent (in log-time) before the prefix vector is fully computed in the leaves. The base case is evident from the definition of Leaf(k).

Richard King observed that, while the solution above executes in the specified time bound, a minor variation (equally expressible) derived by his closure-transformation method has an additional advantage that it can be pipelined (yielding m prefix summations of size n in time m + [c × (lg n)]); the above device cannot because its last leaf remains busy adjusting its sum for lg n steps.

8.5.2 Hardware Corollaries of KTL Constructs

Procedures. The view that a procedure-invocation is an assertable event is commonplace at the hardware level: invocation corresponds to placing arguments in input ports and then asserting a "start" signal to the hardware module that does the computation; a "done" signal indicates availability of results at the output ports.

Actions. In order for a computation unit to be atomic, it must have exclusive access to resources (registers, memories, busses) that it writes, and shared or exclusive access to resources that it reads; it must not release access to a resource until it has finished all uses of it. These constraints rule out internal communication with external processes during the action. Mutual exclusion (resource locking) is ultimately based on bistable hardware devices; see **[Lamport 86]** for a careful analysis of mutual exclusion and arbitration at both software and hardware levels.

Communication through a one-slot buffer or *port* is a typical case. Two Boolean semaphores, one for exclusive access(while reading or writing), another to record the state (full or empty), suffice to control access to a port.

CAM Hardware. That *simple* guarded commands sometimes have hardware realizations has already been noted. Something else that should be noted is the fact that existentially guarded commands provide a natural interface to *content-addressable memories*. Consider a persistent ternary predicate variable Prop. Given that (Prop o a v) is asserted, $\underline{\exists\ \text{z(Prop o a z)? A::C}}$ finds z, $\underline{\exists\ \text{x.(Prop x a v)?A::C}}$ finds o, and $\underline{\exists\ \text{x,z.(Prop x a z)?A::C}}$ finds <o,v>.

If Prop is a persistent mail predicate variable then repeated activations of one of the queries above will receive and retract all relations that satisfy it. Alternatively, given that ∃ w.G?A;;C means A;;C happens for some w such that G, we can introduce an "enumerate for all" construct

$$\forall\ \text{w.G?A;;C}$$

that concurrently finds all w such that G holds and does A;;C for each. Then we could scan through each solutiion for the above queries regardless of whether Prop has the mail or the broadcast property. Some of these operations have fairly direct CAM hardware realizations; it might be appropriate to specify them at the KTL level in modelling such devices.

Pulses and Latches. Consider the pulsed-event and event subtypes (E and G above). A pulsed-event will activate every interrogative-guarded command that is (momentarily) made true by its assertion. Synchronous parallel computation steps can be driven by such communications. An event acts as a latch, remaining true once set until some task resets (retracts) it. External pulse-like events are converted to event variables by a hardware latch known as a Schmidt-trigger.

Alternative-selection. Case constructs are ubiquitous in hardware. Often the guards can be evaluated in parallel; the first-guard satisfied selects its corresponding command and suppresses evaluation of the others. A bounded-arity alternative-select command might therefore be executed efficiently by n closely coupled communicating processors, each receiving one of the alternatives. The same processor structure could be used for closely coupled concurrent tasks (below). The suggestion is not inconsistent with RISC architecture; relations between closely coupled parallelism and RISC architecture remain to be explored.

Concurrency is often realized by parallelism in hardware. In cases where it is not there may be hardware support for priority queues. Again, there is need both for knowledge-based compilation of concurrent compositions to hardware, and for synthesis of closely coupled processor structures that can efficiently process concurrent compositions and alternative-selections in software.

Recursion. General recursive commands imply stacking of control and data structures; hardware support for stacks has long been available. Equally important is the tail-recursive form, which cleanly represents a variety of iterative constructs without use of "exit" commands (it's the tail recursion that is explicit; exit occurs on the paths that don't recurse). Tail recursion requires no control or data stacking.

More generally the μ-operator defines *rational data structures* , a very useful extension of finite data structures to those that can be represented as finite rooted (possibly cyclic) ordered directed graphs. The most likely beneficial form of hardware suppport for rational object processing would be vertex-marking to detect or prevent cycles during recursive graph processing, and reentrant paths during parallel graph processing. This would be one aspect of a general structured-data access control mechanism. (Current LISP machines do not provide this level of support at the software level; some do at the firmware level.)

8.6. Conclusions and Research Directions

We have indicated how to model parallel computing system architectures in KTL. Progress in several areas is needed to extend this work to a useful synthesis basis for such systems. An effective parallel computing system synthesis system will require further progress in several related areas that are being explored under various contracts:

- completion of a KTL-to-CommonLISP compiler and supporting environment for rapid-prototyping of parallel computing-system architectures. We need to extend concurrent composition with communication primitives as described above, and we need to extend the compiler for abstract-object type definitions.

- inference-support for specification, system state and transformation validation.
  We need to develop a basic capability for reasoning about temporal-logic specifications, both for consistency analysis/maintenance of specifications and for validating transformations.

- further work on architectures that support efficient execution of KTL constructs. Most of these constructs are shared with other concurrency and symbolic computation languages. Support for communication in tightly coupled (e.g., multi-RISC) processing systems and parallel evaluation of alternative-selection guards, concurrent compositions are appropriate areas to investigate.

- development of the software-hardware correspondence knowledge base. Each synthesis task must be supplemented with a specification of which software constructs are to be considered "primitive" in the sense that they can be directly executed or compiled to VLSI hardware. Such a knowledge base would specify that , e.g., certain existentially-quantified communications correspond to retrievel from an associative communication-relation store, or that clocked execution is modelled by synchronous-broadcast signals.

- development of transformational synthesis knowledge packs for specific problem areas. These areas may be somewhat general; e.g., a synthesis-knowledge pack for processor trees would contain "divide and conquer" transformations and axioms characterizing conditions under which the resulting processor tree can execute with specified performance constraints.

- exploratory work on targeting synthesis to the VHSIC Hardware Description Language. This work would lead to the capability of synthesizing architecture specifications in a form that can be given to VLSI compilers being developed by other research groups.

REFERENCES

[HG 86] Hillis, W. Daniel and Guy Steele, Jr. *Data Parallel Algorithms,* CACM (Dec. 1986),1170-1183.

[Moszkowski 86] Moszkowski, Ben, **Programming in Temporal Logic**. (Cambridge University Press, 1986)

[Shapiro 86] Shapiro, Ehud, *Concurrent Prolog: a Progress Report,* IEEE **Computer** (Aug. 1986), 44-59

[Gelernter *et al*] Gelernter, David *et. al.*, *Linda and Friends*, IEEE **Computer** (Aug. 1986), 26-34

[Green *et al*] Green, Cordell, David Luckham, Robert Balzer, Thomas Cheatham, Carles Rich, **Report on a Knowledge-Based Software Assistant**, KES.U.83.2 (June 1983)

[Pnueli 86] Pnueli, A. *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: a Survey of Current Trends* , Current Trends in Concurrency, LNCS 224 (1985), 510-584

[Goguen] Goguen, Joseph, *Eqlog* (SRI Technical Report, 1985)

END
7-87
DTIC